NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

THE FORENSIC POTENTIAL OF FLASH MEMORY

by

James E. Regan

September 2009

Thesis Advisor:	Simson Garfinkel
Second Reader:	George Dinolt

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE The Forensic Potential of Flash Memory			**5. FUNDING NUMBERS**
6. AUTHOR(S) James E. Regan			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis explores the forensic opportunities afforded by flash memory. It starts with a discussion of flash storage starting with the physics of flash devices, the development of flash translation layers (which allow flash devices to be used with unmodified legacy operating systems), and flash file systems (which provide for better utilization of flash storage at a somewhat higher cost). Then this thesis provides a comprehension survey of the relevant academic literature and evaluates the work that others have done in the field of flash data recovery. It provides a theory of circumstances when residual data may exist on flash memory through the intentional deletion and overwrite of previously saved data, based upon a thorough patent review and freely available documentation. It clearly documents the steps of configuring a Linux kernel to use the YAFFS2 (Yet Another Flash File System used in Android) and the JFFS2 (the Journaling Flash File System used on the One Laptop per Child Program) flash file systems. It then conducts experiments to confirm or deny these theories, with a focus on the recovery of data and other evidence that overwritten and deleted data once existed. Finally, this thesis makes recommendations for further research.

14. SUBJECT TERMS Flash Memory, Forensics, Flash File Systems, Flash Transition Layer, YAFFS, JFFS2			**15. NUMBER OF PAGES** 99
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

THE FORENSIC POTENTIAL OF FLASH MEMORY

James E. Regan
Captain, United States Marine Corps
B.A., Franklin and Marshall College, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: James E. Regan

Approved by: Simson Garfinkel
Thesis Advisor

George Dinolt
Second Reader

Dr. Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis explores the forensic opportunities afforded by flash memory. It starts with a discussion of flash storage, starting with the physics of flash devices, the development of flash translation layers (which allow flash devices to be used with unmodified legacy operating systems), and flash file systems (which provide for better utilization of flash storage at a somewhat higher cost). Then this thesis provides a comprehension survey of the relevant academic literature and evaluates the work that others have done in the field of flash data recovery. It provides a theory of circumstances when residual data may exist on flash memory through the intentional deletion and overwrite of previously saved data, based upon a thorough patent review and freely available documentation. It clearly documents the steps of configuring a Linux kernel to use the YAFFS2 (Yet Another Flash File System used in Android) and the JFFS2 (the Journaling Flash File System used on the One Laptop per Child Program) flash file systems. It then conducts experiments to confirm or deny these theories, with a focus on the recovery of data and other evidence that overwritten and deleted data once existed. Finally, this thesis makes recommendations for further research.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Small devices using flash-based storage are routinely encountered during military, law enforcement and intelligence operations. Today, these devices are largely analyzed using tools designed to analyze hard drives. These tools look for data at the logical block level. They can find data inadvertently left on the device either because no attempt was made to delete it or the attempt was not successful. Flash devices offer additional opportunities for data recovery, because there is a physical layer below the logical layer that can be exploited to benefit the forensic investigator.

This thesis explores the forensic opportunities afforded by flash memory. It begins with a discussion of flash storage starting with the physics of flash devices, the development of Flash Translation Layers (FTLs), which allow flash devices to be used with unmodified legacy operating systems, and flash file systems (which provide for better utilization of flash storage at a somewhat higher cost). The thesis then makes the following contributions to the field of computer forensics:

- The first comprehensive survey of the academic literature regarding flash forensics.
- A thorough review of FTL, flash file systems and flash memory patents with respect to the opportunities for recovering residual data.
- Clearly documenting the steps for configuring Linux to use YAFFS and JFFS2 with a flash simulator.
- Performing experiments that used a flash simulator and file system operations to determine residual data left by YAFFS and JFFS2.
- Discussing the possibilities for recovering residual data from a FTL.
- Recommendations for further research.

These techniques further developed the understanding of how residual data is left on flash memory devices and how to recover it.

B. THEORY

Flash is a block structured storage system that is increasingly being used to supplement or replace traditional magnetic storage in many applications. Flash is also widely used on portable devices such as cellular phones. But flash storage is different from magnetic storage in two important ways. First, flash storage blocks must be *erased* before they can be written, and the erasure size is typically much larger than the block or sector size. Second, whereas the sectors on a hard drive can be rewritten hundreds of millions of times, flash blocks can typically only be rewritten a few thousand times.

The physical limitations of flash media are overcome with two technical approaches: out of place writes when data is changed, and wear leveling that swaps data from one physical location to another in order to even the usage of erase operations among the flash storage blocks. This required functionality offers more opportunities for a forensic examiner to recover residual data than are afforded on a typical hard disk drive.

Flash media used in SD cards and in USB thumb drives implement these requirements through the flash transition layer (FTL), an indirection layer that allows a traditional operating system to read and write individual logical sectors on the flash, but which translates these operations to the out-of-place writes and wear-leveling necessary for proper operation. As a result of the FTL, the act of overwriting every logical sector of a flash storage device may leave observable evidence at the physical layer that the flash media has been previously used. This is because overwriting blocks at the logical level will result in changes to the internal FTL data structures that cannot be controlled through the traditional APIs used to control block devices. It is also possible that residual user data may be recovered through an API that allows access to the physical flash device, because the act of overwriting specific logical data does not translate into the erasure and overwriting of the corresponding physical blocks.

The potential for forensic recovery with flash is therefore greater than with a hard disk drive. Users of hard drives have full access to the physical layer: the act of wiping the entire hard drive and reinstalling a new operating system leaves little or no trace that the device had been previously used. But, if a user were to reinstall a new operating

system on a flash device in attempt to cover previous use, the data structures are written to new physical locations, possibly leaving older ones in place, unmapped to the logical layer. These unmapped, older blocks may contain previous operating system data or user files that will provide a signature different from a flash device that was used just once.

In addition, a flash device used for the first time will most likely result in very little fragmentation at the physical layer. On a clean device, data is written sequentially until the media fills. There is very little wear leveling if the media has not been used much. But, if the flash device has been used extensively in an attempt to wipe at the logical layer only, the reinstallation of an operating system will result in a greater level of physical fragmentation as the media fills with out of place writes and the FTL or flash file system conducts wear leveling procedures.

This theory is also applicable for use in the potential recovery of previous user files that a user attempts to wipe. But, as the user files grow in size and the use of the flash media increases, the files will become more fragmented at the physical layer, leaving data recovery operations much more difficult. Smaller files, particularly the size of a physical flash page or smaller (approximately 512KB), should provide the greatest potential for recovery. The ideal targets for recovery would be small text documents, cell phone message log entries, contact information, small images and file system data structures that contain metadata such as inodes and FATs. Files that are likely to become fragmented and harder to recover include large image files, video files and sound format files such as .mp3 and .wav.

C. FLASH TRANSLATION LAYER

There are two approaches to address the limitations of flash memory caused by write endurance, out of place writes, erase block sizes and NAND flash's non-random addressability. The first is the Flash Translation Layer specification. The FTL allows any standard file system to utilize a flash memory device by emulating a block device. The FTL stands between the host operating system and the flash device and translates the standard block commands from the host. It will present memory from the flash in sectors and blocks, much like the standards used by a hard disk drive interacting with a FAT file

system, while hiding the intricacies of the flash device, such as block erasing and wear leveling. It does this by mapping the block and sector addresses of the standard file system to physical addresses. So, as data is physically moved to different locations in order to implement wear leveling and out-of-band writes, the FTL presents the data to the host operating system as if the data is written to a static location. The FTL may be implemented on a hardware controller that exists on the chip itself working as the intermediary between a host OS utilizing a standard file system such as FAT or it may exist as part of the operating system. A common example of an implementation is SanDisk's mobile flash drive, which utilizes an FTL programmed on an embedded controller within the SanDisk SD memory cards.

D. FLASH FILE SYSTEMS

The second approach is to design a file system that implements wear leveling and out-of-band writes itself. Applications such as the open source YAFFS (Yet Another Flash File System) and JFFS (Journaling Flash File System) and the proprietary Microsoft Flash File System use this approach. Instead of using the traditional FAT or inodes, these systems create their own data structures on the flash device. These data structures are loaded into the host computer's RAM when the flash file system is mounted. These structures are used by the file system to implement wear leveling, bad block management, block reclamation during block erasure and pointers to extents that will recreate files requested by the host. Typically, the structure will utilize status flags that indicate the state of the physical pages and blocks and erase counts to even wear. A key distinction between FTLs and flash file systems is that the operating system, through the flash file system, will have direct access to the spare area to store metadata and file structures, whereas the FTL does not give the OS and utilized file systems that level of access. Flash file systems are typically used in embedded applications where the flash storage is not removable, such as the YAFFS implementation in the Android operating

system for mobile phones [1].[1] Unlike the FTL utilizing a FAT, flash file systems are not as portable to other systems; specific drivers will need to be loaded onto the host for interaction.

The potential for forensic data recovery will depend upon the specific implementation of the wear leveling mechanism. Each specification will handle wear leveling differently and will perform garbage collection at different intervals. One solution will not fit all, and each system needs to be studied carefully to maximize data recovery.

[1] The T-Mobile G1 phone, which uses the Android operating system, actually uses both a flash file system and an FTL. YAFFS is used for the phone's internal flash storage and the Microsoft FAT32 file system with an FTL to store data on the phone's micro SD card.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND RELATED WORK

A. BACKGROUND

1. Physics of Flash Memory

Flash memory is a type of Electronically Erased Programmable Read Only Memory (EEPROM). Flash can be in one of two states—erased and non-erased. Flash is nonvolatile, in that it retains its content after the removal of power. Flash memory cells are made up of floating gate transistors to store information, where the gate traps an electron. The existence of a charge indicates a zero and no charge represents a one. Write operations can only program a one to a zero. In order to clear a bit (change the value of a bit to one), an entire block of memory must be erased.

A flash cell is made up of a floating gate, which is a transistor that is completely surrounded by insulating material and is governed by a control gate. A process known as channel hot electron injection causes an electron to gain enough energy to pass through the isolating material. The electrically isolating property allows the floating gate transistor to then trap the electron. The electrical charge that is applied to the gate comes from the bit line at the drain side. A positive charge, the absence of a trapped electron, is associated with the logical one while a negative charge, electrons trapped in the floating gate, is associated with the logical zero. The only connection the floating gate has to the row, or word line, is through the control gate. The trapped electrons give the floating gate a negative charge and work as a barrier between the floating gate and the control gate. When the charge passing through the gate is measured and it is above a specified threshold, consistent with no electrons trapped within the floating gate, a one is indicated on read. Conversely, when electrons are trapped, the charge passing through the gate should drop below the threshold and a zero is read. The insulating material is what gives the flash its non-volatile property; the electron is trapped without the need of a constant refresh. Programming NAND cells must be done at 512 byte page intervals, while NOR cells can be programmed on a word basis [3].

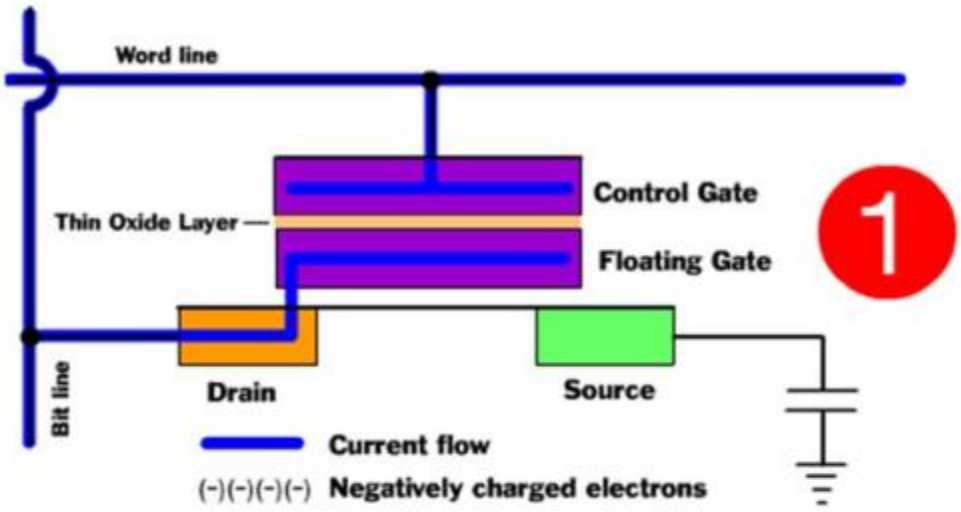

Figure 1. Flash Cell Erased State (From: [4])

Erasing a flash cell utilizes the Fowler–Nordheim tunneling process to remove the electrons from the floating gate. A high voltage of opposite polarity is applied to the floating gate, forcing the trapped electrons through the insulating material to the surface. The negatively charging barrier is thus eliminated and the charge passing through the gate will rise above the threshold [3].

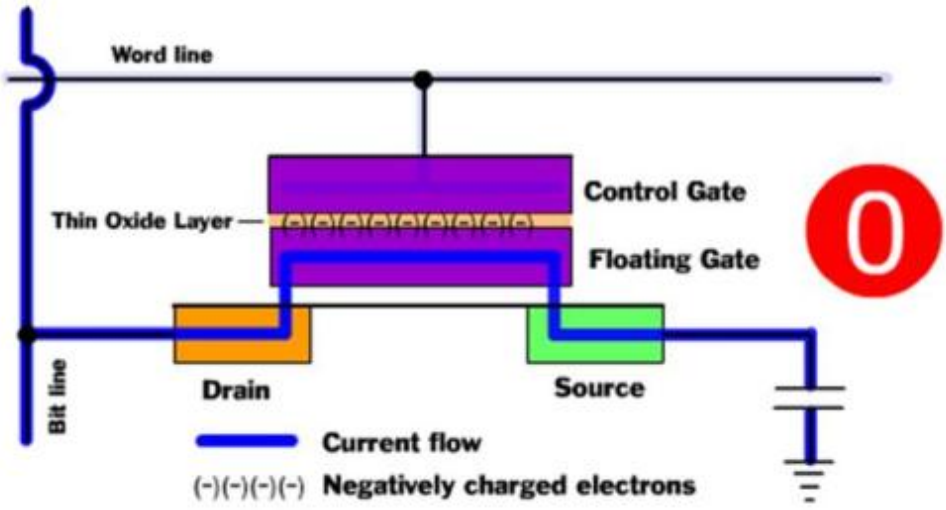

Figure 2. Flash Cell Programmed State (From: [4])

Flash cells can be Single Level Cells (SLC) that hold one bit per cell or Multi Level Cells (MLC) that can store more than one bit per cell. Storing more than one bit per cell is accomplished by storing multiple possible levels of electrical charge and using multiple threshold voltage levels. Typically, MLC may represent four different states [5]. The tradeoff in more capacity of the MLC flash is increased read/write latency and a shorter lifetime [6]. The write latency is due to a more sophisticated programming technique to store the precise charge needed to achieve the threshold voltage distribution. Similarly, the read operation is longer because it takes longer to distinguish between the four possible charges stored in the cell. The lifetime of MLC devices decreases due to the inability to distinguish between the four different states, compared to only two in SLC [7].

2. History and Trends of Flash

The patent for NOR flash memory was filed by Fujio Masuoka in 1981. Masuoka was working for Toshiba in Japan and was able to produce the first chip in 1984. By 1987, Masuoka developed the first NAND flash chip. NOR flash was developed to store smaller amounts of data, have low access latencies and be able to execute in place, avoiding having to load software into RAM. It was ideal for BIOS and firmware, information that rarely needs to be updated. But NAND was developed with the idea to replace the hard disk drive market as long term storage. The first NAND devices went on the market in 1990 [8].

With its small size, low power consumption, storage density, shock resistance and low cost, compared to other EPROM, NAND flash memory is the choice when solid state non-volatile memory is needed, especially in mobile devices. As demand for personal devices such as digital cameras, cell phones, portable video game consoles and music players have grown, so has demand for flash memory. The oldest applications date back to the mid 1990s and included SanDisk's CompactFlash, Toshiba's SmartMedia, Siemen AG/SanDisks's MultiMediaCard and the physically larger Memory Stick introduced by Sony. By 2000, Panasonic, SanDisk, Toshiba, Kodak developed Secure Digital cards and these devices have been the dominant removable format of flash memory in portable

devices. But SD cards have not penetrated the embedded system market because of their lack of compatibility with IDE/ATA; CompactFlash has had better market penetration. Various manufacturers have also produced USB flash drives since 2001.

Memory chip capacity roughly follows Moore's law because they share the same equipment and techniques used in production of integrated circuits. Flash prices consistently dropped 30 – 40% from the late 1990s through 2003, but NAND flash has accelerated to a 50% decline driven by increased supplies. Prices were approximately $10 per gigabyte in 2007 and have dropped to approximately $2 per gigabyte in 2009 [9].

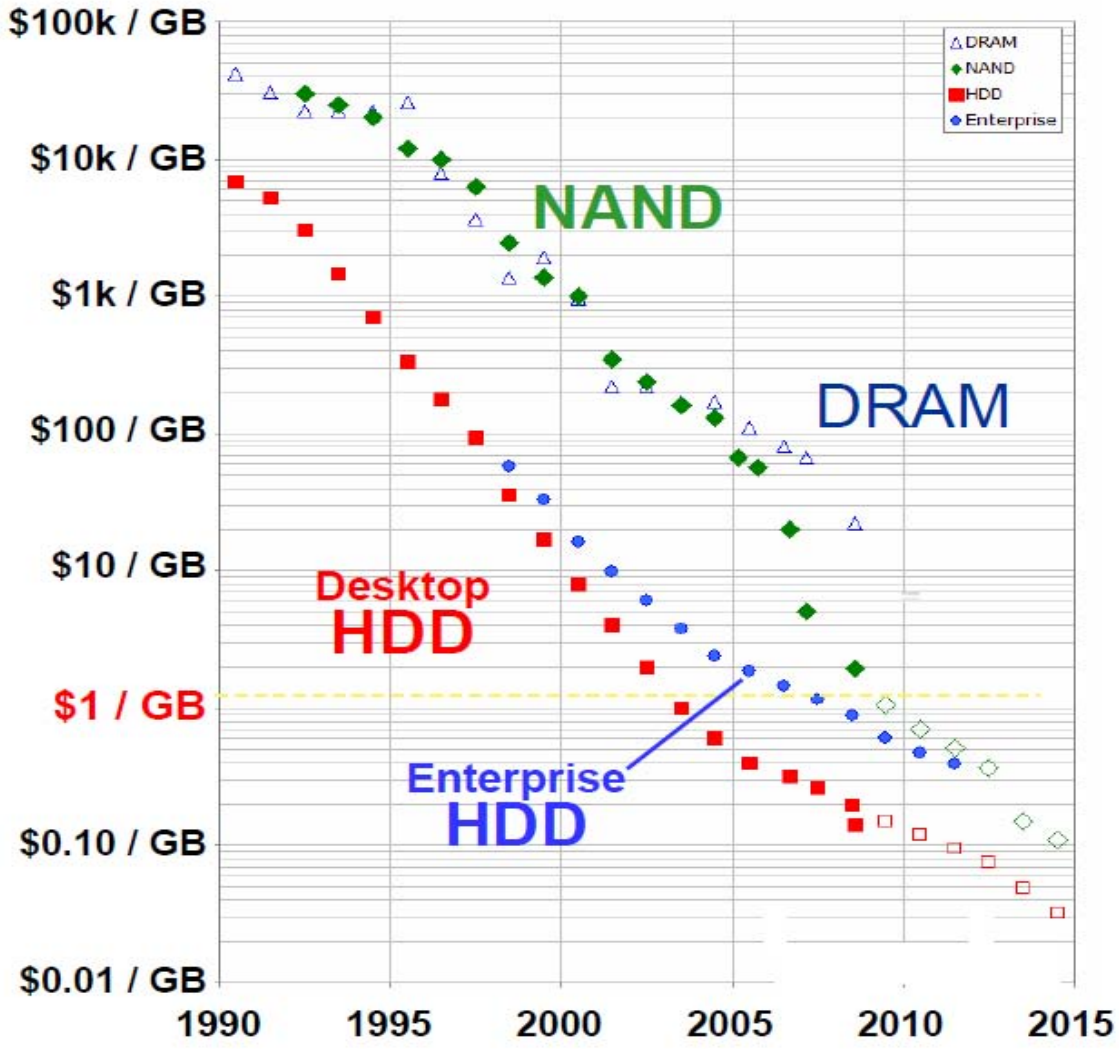

Figure 3. Price/GB Memory Trend (From: [9] with permission)

3. NOR Versus NAND Flash

There are two types of cell arrays in flash memory, NAND and NOR. They differ in how the memory arrays are connected and how memory is addressed to perform a read or write operation. The cells in a NOR flash device are connected in parallel, allowing each cell to be read and programmed individually. This connection resembles a NOR gate. Read addressability is much like Random Access Memory: NOR flash can be read byte by byte in constant time. In NAND flash, the cells are connected in series, much like a NAND gate, which prevents an individual cell from being read or written to. Therefore, one complete interconnected series may be read or written. NAND flash uses a shared bus for addresses and data transfers, while NOR uses a separate BUS for addressing, memory write and read operations [3].

The tradeoff from lack of cell level addressability in NAND flash is increased density. NAND flash technology devices are more economical per bit and can store more information in a smaller area than NOR flash devices. NOR flash was invented as an economical replacement for ROM, while the increased capacity of NAND was envisioned as competition for Hard Disk Drives as a secondary storage device. One other advantage NAND has over NOR is a faster erase time. This improvement is recognized through a smaller erase block. The typical erase block size in NOR ranges from 64 to 128 Kbytes, while the typical NAND erase block size ranges from 8 to 32 Kbytes. The smaller NAND block size will speed an erase operation from 5 seconds in a NOR device to 4 milliseconds in a NAND device [10].

Because of its serial nature, NAND flash has a multiplexed input/output bus that carries both address information and data. The bus is typically 8 or 16 bits wide and is too small to carry an address in one cycle. Data is accessed by the address of the data first applied in three to five cycles. Once the address is loaded, the same input/output line is used to transfer the data residing at that address [11].

4. Flash Endurance and Limitations

Flash has a limit to the number of erase-write cycles. The upper limit varies greatly from sources, but the range is typically 10^4 to 10^6 writes and the limitation is

improving with time. This limit is also referred to as write endurance. SLC flash typically has a higher write endurance than MLC and NOR generally has a better write endurance than NAND flash [2].

Manufacturers may implement a *wear leveling* scheme in order to even out the wear across all flash cells and increase the effective lifetime of the capacity of the device. Wear leveling will not improve the lifetime of any individual cell; rather, it spreads write operations across all cells, so no one particular cell wears out before others, thus reducing the capacity of the device.

Once bits in a block can no longer be erased and permanently hold a value of zero, the block is marked as bad. NAND flash devices are shipped with bad blocks already existing on the device. Typically, 2% of a NAND flash device will contain bad blocks when shipped [11].

In addition to write endurance, there is an upper limit to how long flash memory can hold data. The quality of the voltage level will deteriorate over time. Most data sheets for commercially available flash indicate an upper limit range of 10 to 100 years before data is lost [12].

5. Flash Memory Logical Structure

The logical structure of flash memory from least to greatest granularity is erase zones, blocks and pages. Erase zones usually consist of 256 to 1024 blocks. Not every device uses the concept of erase zones. One or more erase blocks may be organized into an erase zone. A logical concept, zones can be used to manage bad blocks. As blocks go bad, data may be swapped into good blocks from the bad blocks within the same zone.

The flash block is the lowest unit that may be erased. A block may consist of 32, 64 or 128 flash pages. The total number of blocks per flash device varies upon the total storage capacity. Flash pages are usually a multiple of 512 bytes of usable storage area (excluding spare area) and typical sizes include 512 bytes, 2K bytes or 4K bytes. The flash page is the lowest addressable unit in NAND [11].

Flash pages are further comprised of usable area and spare area. The usable area is where user data is stored and the spare area is used for flash metadata. The spare area size can range from 8 to 64 bytes, depending on the size of the page. The metadata will contain information regarding the page number, which erase block the page belongs, and will be used to map the physical location of the data back to the logical structures of the flash transition layer. Other information held within the spare area could include a dirty bit, which will mark a particular page as outdated, indicating that the information has changed and that the page no longer contains the most current data and ECC data. It may also contain a bit that marks the block as ready to be erased and if the block has gone bad (no longer usable because, for example, it has reached its end of lifetime due to the number of erase operations performed). The particular structure and information contained within the spare area is not standardized and is manufacturer specific. This information will be used by the wear leveling algorithms and as such will be proprietary. The spare area may be used to describe an individual page or they may be aggregated to describe an entire erase block [11].

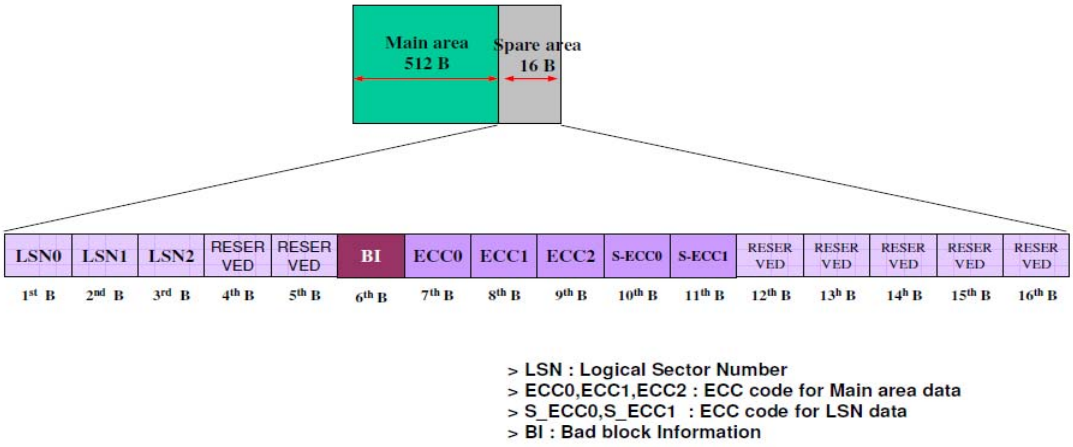

Figure 4. Example of Spare Area Detail (From: [14])

6. Flash Specific Operations

A flash page may be individually read and programmed (bits changed from one to zero), but an individual page cannot be erased (set to one). For example, a byte may be programmed from 1111111 to 11111100. Further possible changes may be 11111000 or

11011000, but 11111101 is not possible, as it changes the rightmost bit from a zero to a one. Flash allows random access reads and programs, but not rewrites and erasure operations [2].

There are three possible operations that may be performed on a flash memory device. They are read a page, program a page, and erase a block. The read operation is the fastest of the three and typically takes 25 microseconds for NAND flash and 20 nanoseconds for NOR. A program operation will typically take 250 microseconds for NAND and 10 microseconds for NOR. The erase block operation is the longest and may take as long as 3 milliseconds for NAND and up to 5 seconds in NOR [5] [12].

To maximize the lifespan of the memory cells, the manufacturers of the flash implement a wear leveling algorithm. In contrast, a magnetic disk in an overwrite operation will reuse the same sector that the data was originally stored upon. So as data is written, deleted and rewritten, the early sectors on the disk will be used much more than the later. Wear leveling is not as large a concern to the manufacturers of magnetic storage disks, because the ceiling of the lifetime of the hard drive is not dictated by the lifetime of the sector, but rather by the mechanical device needed to write and read data off the disk. But if the operations used for a magnetic hard disk drive were used on flash, blocks used to hold file system metadata (which change often) would wear out and go bad. This would decrease the overall effective lifetime of the device [14].

The algorithms used for wear leveling spread the write and erase operations over the entire flash device, so that wear occurs evenly. Further, when data is changed, the same page is not reused, but the updated data is written to a new physical page, that is set to all ones and the old page will be marked for erase. This is referred to as an out-of-place write. In order to maximize power and wear leveling efficiency, a block is not erased immediately but rather a group of blocks will be erased when space is needed [14].

7. Access via the Joint Test Action Group (JTAG) Interface

Obtaining a logical acquisition will not create a complete picture of all the data stored on a flash device. A complete image of the physical layer will produce all stored data. One non-invasive method of obtaining physical access to flash memory chips is

through the Joint Test Action Group (JTAG) connection pads [13], also known as the "Standard Test Access Port and Boundary-Scan Architecture." A JTAG access port is normally used to test selected printed circuit boards during the manufacture process and to debug embedded software, but can also be used to obtain a physical layer image of flash memory chips, indirectly through the circuit board. JTAG access ports do not physically exist on flash memory chips, but are built into the circuit boards that are connected to flash memory chips.

The JTAG approach is not without its problems. The location of JTAG ports are not always published by the manufacturer of embedded devices and the ports do not exist on all devices [13]. Some devices that have JTAG ports do not give the interface unrestricted access to the flash memory map. Finally, the JTAG ports may be disabled after production prior to distribution to the end user.

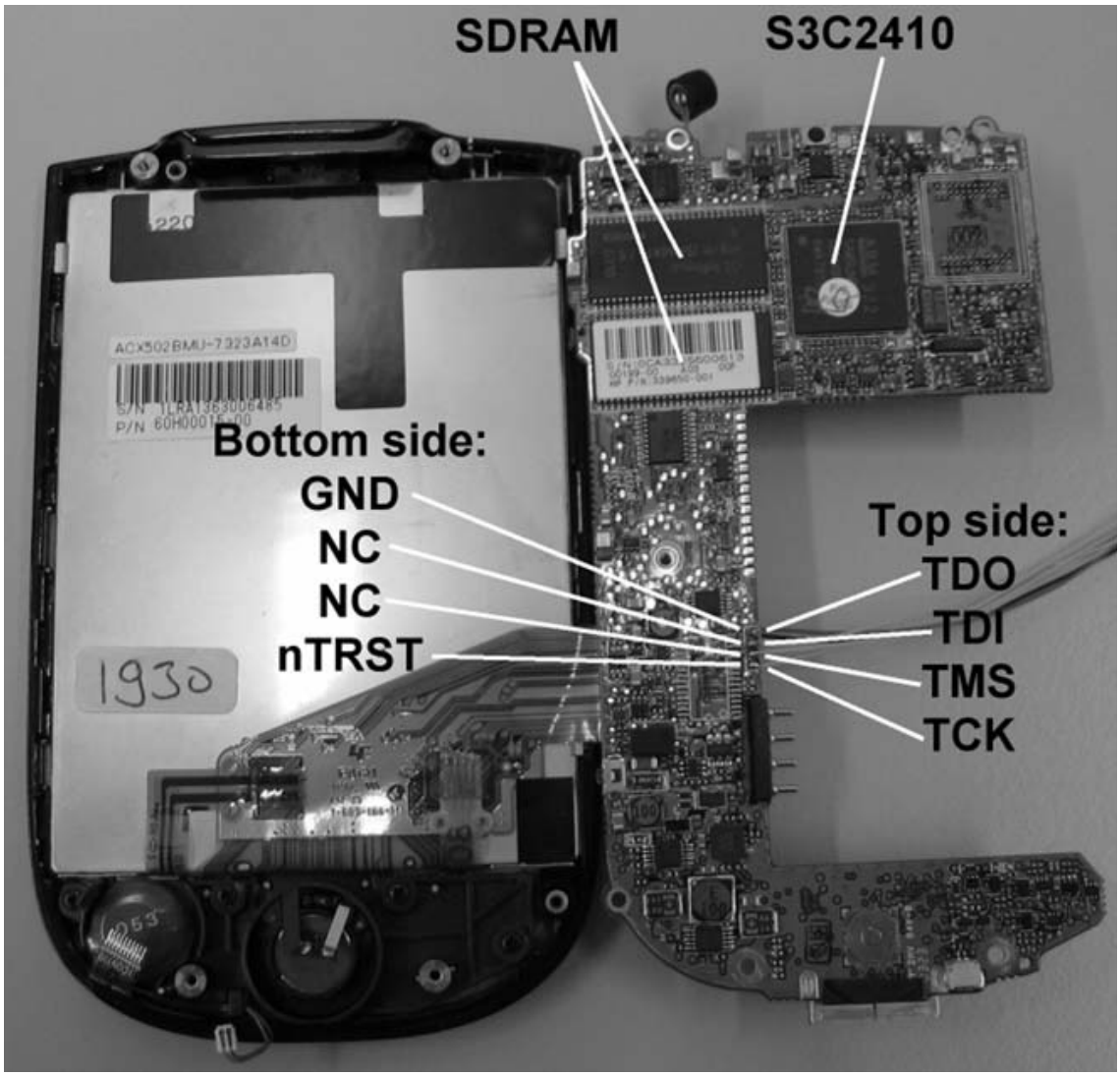

Figure 5. JTAG Ports on a Flash Embedded Device (From: [13] with permission)

8. Wear Level Approach with the Flash Translation Layer

The purpose of the FTL is to present flash memory to a file system as if it were a rewriteable block device, such as a magnetic disk, while the device driver, or controller hardware, addresses the peculiarities of flash memory that the higher level software is not concerned with, such as wear leveling, block reclamation and crash recovery. The basic idea behind the wear leveling technique is to map the virtual block number presented by the host operating system to a physical flash page on the memory chip. There are no known open source FTL implementations.

FTLs are complex because the map is stored on the flash device itself, with only a small portion of the map in memory. FTLs must track which logical blocks are updated most frequently in order to minimize the cost of updating the map when the physical location of the data has changed. Different FTLs have been implemented by different manufacturers. There is no need for compatibility between FTLs, because most exist in standalone storage devices such as USB thumb drives, SD cards and solid state storage devices [2]: compatibility is provided at the logical layer. This lack of compatibility would surely complicate any forensic analysis that took place beneath the FTL at the physical level.

Intel's FTL implementation consists of two mappings: a virtual block number to a logical block and page number map and then a logical block to a physical erase block map. Most of the virtual block to logical block map is stored on the flash itself, except for the first few virtual block entries, which change often in a FAT file system. The logical block numbering scheme minimizes map changes, for when valid pages are rewritten to a new physical block, the page offset into the new physical block remains the same and the logical block number need not change. On the flash, there is also a secondary virtual to logical map. This secondary map is used for efficiency, for changes to the map can be written to the same physical block, delaying a block reclamation operation, while the primary map entry is marked as obsolete [15].

The second map is a much smaller logical block to physical block mapping that is loaded into RAM when the flash memory is mounted. This mapping changes often, but the cost is much less as it resides in RAM [15].

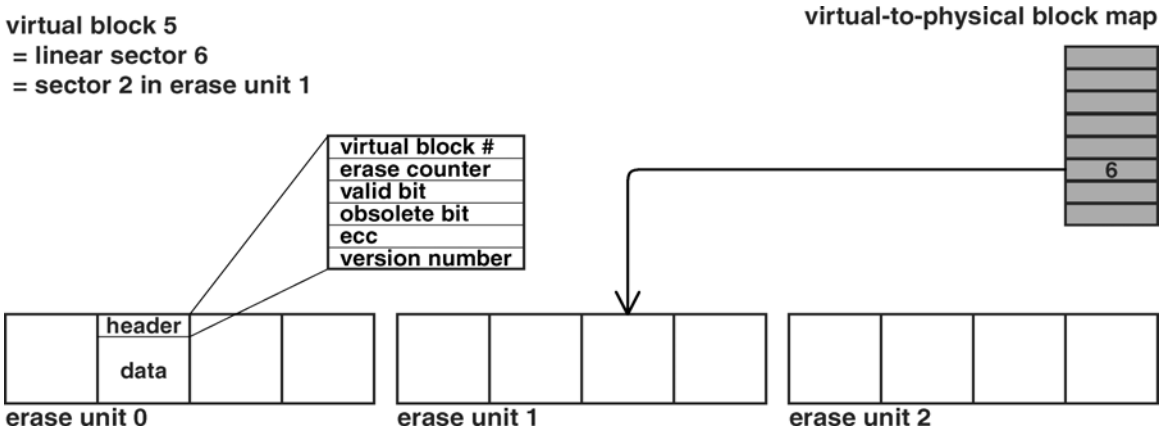

Figure 6. FTL Mapping Structures (From: [2])

Other FTLs include the NAND FTL, which uses one of two different mapping schemes, depending on whether the flash has spare area. The mapping scheme associates a chain of physical blocks with a virtual block. If the flash does not have a spare area, the NFTL searches the chain sequentially until it finds the relevant valid page associated with the virtual block. When data is changed, the page is written to the next physical block and the old block becomes obsolete. Once all blocks in the chain are used, all valid pages are written to a new block and it becomes the head of a new chain. If the device uses spare area, the chain is no longer than two. Changes to pages are written to the second physical block in the chain, the backup block. There may be multiple versions of virtual blocks in the backup physical block, but when it is full or space needs to be reclaimed, all valid data is written to a new physical block, which becomes the head of a new chain of length two and the old chain of physical blocks are erased [2].

Devices utilizing Windows CE 2.1 and later, along with in-line flash memory, or flash memory without a controller on board, implement M-Systems' TrueFFS as a flash translation layer [16]. For the release of Windows CE .NET 4.2, Microsoft claimed that it would no longer support TrueFFS [17]. TrueFFS is disclosed in U.S. patent number 5,404,485, but the details of the format used to store relevant data in the spare area, such as markers for invalid pages are not included. According to the patent, each erase block contains a header and a map for each page within the block. But the patent does not specify where, in the block, the map is stored; whether it is distributed amongst the spare

areas for each page or contiguous at the start of the block. Each map entry for a particular page will have a status field for its corresponding page; indicating whether the page is free and writeable, deleted and not writeable, or allocated and holds user data along with its logical address [18].

9. Wear Level Approach with Flash File Systems

While the Flash Translation Layer hides the management of flash memory to the file system by disguising it as a block device, a file system that is designed specifically for flash exposes all the details and lets the file system manage wear leveling and block reclamation. The advantage of the flash file system solution is that it can be more efficient as the file system is not buried beneath layers of mapping used for block device emulation. This makes it ideal for resource-constrained devices such as cell phones. But the FTL is a better solution for removable media, as the FAT format is a rewriteable file system understood by all Windows, Mac and Unix/Linux platforms [2].

Examples of flash file systems include JFFS/JFFS2 [19] and YAFFS [20]. JFFS was created for a specific application and is not widely used. JFFS2 was used on the One Laptop Per Child program [21] and YAFFS is currently being used as the file system for embedded flash in Android [1]. Unlike the FTL, JFFS/JFFS2 and YAFFS are open source.

Yet Another Flash File System was developed by Aleph One as a file system designed specifically for NAND flash. YAFFS stores its file data in chunks. Each YAFFS chunk is the same size as a NAND flash page, 512 bytes. YAFFS stores metadata, or tags, in the spare area. This information contains a file id number, a chunk number, a write serial number, a tag error correcting code, a page status field, a block status field and the bytes-in-page used. It is the same format used by SmartMedia on PMCIA cards [20].

Byte	SmartMedia Usage	YAFFS Usage
0…511	Data	Data. Either file data or file header depending on tags.
512…515	Reserved	Tags
516	Data Status Byte. Not used.	Data Status Byte. If more than 4 bits are zero, then this page is discarded.
517	Block Status Byte	Block Status Byte
518…519	Block Address	Tags
520…522	ECC on second 256 bytes of data	ECC on second 256 bytes of data
523…524	Block Address	Tags
525…527	ECC on first 256 bytes of data	ECC on first 256 bytes of data

Table 1. YAFFS Spare Area Detail (From: [20])

Field	Comment	Size for 1KB Chunk	Size for 2KB Chunk
blockState	Block state. non-0xFF for bad block	1 byte	1 byte
chunkId	32-bit chunk Id	4 bytes	4 bytes
objectId	32-bit object Id	4 bytes	4 bytes
nBytes	Number of data bytes in this chunk	2 bytes	2 bytes
blockSequence	sequence number for this block	4 bytes	4 bytes
tagsEcc	ECC on tags area	3 bytes	3 bytes
ecc	ECC, 3 bytes/256 bytes of data	12 bytes	24 bytes

Table 2. YAFFS Tags Usage (From: [20])

The file id is used to associate the chunk with a file. The chunk number is a count that starts at zero, which represents the file header, and increases by one for each chunk needed to store the entire file. A file id of zero indicates that the chunk is deleted, but not yet erased by the flash garbage collector. The block status field indicates whether a block has gone bad and the page status field indicates if the page is valid or discarded. If four bits or more are zero, the page is discarded [20].

When a chunk is "overwritten," meaning that the data has been updated, the new data is written to a new flash page, but with the same tag data. The exception is that each tag receives a 2-bit write serial number that is increased with every write. This serial number contributes to recovery after a system crash during a write operation. If power is

lost after the new chunk is written, but before the old chunk is marked as dirty, the higher serial number will indicate the latest version. The old chunk is then marked as a dirty page within the spare area and is ready for garbage collection.

A block can be erased on two different occasions. The first is if all pages within a block are dirty, and the second is if there is one page that is still valid. In the second case, the valid page will be rewritten to a new block and the current block will now be released for collection [20].

The Journaling Flash File System was designed by Axis Communications and was released for the 2.0 Linux kernel. The motivation behind developing JFFS was to address the lack of wear leveling and unsafe rewrite operations when standard file systems were used, while flash memory was treated as a standard block device with 512 byte sectors. When a standard file system requests blocks from the free list, it tends to favor blocks that hold data which changes often, while ignoring static ones. This will lead to uneven wear with some blocks never changing and others with a high turnover rate [19].

The design of JFFS is a log based file system, in which a circular data structure is used to write all data sequentially in the form of nodes. JFFS writes data linearly through the flash device, where the oldest node is the head and the newest is the tail. The header information for each node contains the name of the file to which it belongs by storing the 32-bit inode number, the name of the file and a 32-bit version number which is totally ordered for each inode. If a node contains data, the header information will also contain the offset in the file where the data belongs. A node with a similar offset, but found earlier in the log is superseded by the later node, distinguished by the version number [19].

Garbage collection works by observing the node at the head of the log. If the node is already obsolete, it is skipped and the garbage collection process moves the head to the next node. If the node is valid, it is rendered obsolete by rewriting the data and header information to the tail and the head is moved to the next node. This process is continued until JFFS has rendered a complete flash erase block obsolete. Because the

garbage collection process rewrites nodes to the tail in order to free a complete flash block, JFFS cannot wait until the entire flash memory has been used—when the tail meets the head. Therefore, a slack space is needed at all times between the tail and the head of the log [19].

In order to provide a data compression capability to meet a customer requirement, the developers at Red Hat reworked JFFS and developed JFFS2. JFFS2 also supports hard links and improves the efficiency of the garbage collection process. The problems addressed in garbage collection also changed the layout of the file directory. JFFS2 no longer uses a circular log data structure. Instead, when the flash device is mounted a map of only essential information is loaded into memory. This map is a hash table which contains an entry for each inode. For each entry in the map there exists a struct, named jffs2_inode_cache, containing the inode number, the pointer to the head of a linked list that contains all the physical nodes containing the file data and the number of links to the inode—to support hard links [19].

Essentially, JFFS2 is a series of linked lists. When a file needs to be accessed, JFFS2 will use the appropriate inode number to look up the jffs2_inode_cache entry in the hash map. It can then reconstruct the physical layout of the file by walking the linked list and deriving the physical location of each node from the entries and where it can remain in memory [19].

All flash erase blocks are assigned to one of three lists: the clean list, which contains blocks that store valid data; the dirty list, which contains blocks that have at least one obsolete node and the free list, which contains blocks that have been successfully deleted. Wear leveling is achieved by statistically choosing the clean list or dirty list to erase a block when room is needed [19].

B. PRIOR WORK

1. Introduction

Prior art in flash forensics can be classified as work that attempts to identify methods to recover remnant data from flash memory, attempts to acquire logical and

physical images, or other, where the authors recognize the peculiarities of flash memory, such as wear leveling and how it impacts data retention. The following is the result of a thorough search of relevant published articles in the field.

Title	Authors	Date
Physical Acquisition		
Forensic imaging of embedded systems using JTAG [13]	Breeuwsma, M	2003
Forensic Data Recovery from Flash Memory [11]	Breeuwsma, M., et al	2006
Logical Acquisition		
Analysis of USB Flash Drives in a Virtual Environment [22]	Bem, D., and Huebner E.	2007
An overall Assessment of Mobile Internal Acquisition Tool [23]	Distefano, A. and Me G.	2008
Remnant Data		
Data Remnants in Flash Memory Devices [12]	Skorobogatov, S.	2005
A Study of Information Privacy and Data Sanitization Problems [24]	Roubos, D., et al	2007
Ten Good Reasons Why You should Shift Focus to Small Scale Digital Device Forensics [25]	Knijff, R	2007
An Integrated Approach to Recovering Deleted Files from NAND Flash Data [26]	Luck, J. and Stokes M.	2008
Recovering data from USB Flash memory sticks that have been damaged or electronically erased [27]	Phillips, B., Schmidt C., Kelly D.	2008
Other		
Algorithms and Data Structures for Flash Memories [2]	Gal, E. and Toledo S.	2005
An Investigation into the Development of an Anti-Forensic Tool to Obscure USB Flash Drive Device Information on a Windows XP Platform [28]	Thomas, P. and Morris A.	2008

Table 3. Prior Work Articles

2. Physical Acquisition

In the paper "Forensic imaging of embedded systems using JTAG (boundary-scan)," Marcel Brueeuwsma proposes using the Joint Test Action Group (JTAG) as a physical means to produce an image of stored data in flash memory on an embedded device. The author's goal is to produce an image of an embedded system with minimal changes to the embedded memory; maintaining forensic integrity. In particular, the author states that for many applications, software needs to be loaded onto the device in order to begin retrieving data. This process results in a loss of integrity. Small-scale embedded devices are a suitable target for this technique because they use memory chips and not disk drives to store data [13].

The JTAG method will make a full forensic copy of the flash memory, but the difficult task is finding the JTAG test pads. The pads are not necessarily labeled on the circuit board and the manufacturer might not publish the locations in other literature. The JTAG pads do not exist on a flash memory chip, but on another component within an embedded device, such as the processor. The authors offer two JTAG modes that will result in a complete image with NOR flash: extest and debug, while only extest will work with NAND. The authors provide a detailed algorithm for identifying JTAG access ports, through a brute force measure of every pad on the chip board (some pads on the board are not JTAG) and two alternative, less reliable and more destructive methods that involve multi-meter testing and x-ray imaging of traces on the circuited board. The process of measuring each pad involves providing an input to the pad and reading the output, matching it with an expectation. Other advantages of a JTAG retrieval include the minimal chance of altering data, while disadvantages include the long process, hard to find ports and the process may require disassembly of the device as not all devices are JTAG enabled [13].

In "Forensic Data Recovery from Flash Memory," Marcel Breeuwsma, et al present a low level, physical layer method to recover data from a flash memory system and then analyze the results in order to rebuild the file system. The authors collected 45 different make and model flash memory sticks in which to acquire data. The authors describe three possible methods to extract the contents from a flash memory chip [11].

The first method is to use a "flasher tool." A flasher is a device used by manufacturers for debugging and performing software updates in the field. Flashers are used by hackers for altering device functionality. Flashers are not generic and no one tool can be used on all devices because of non-standard interfaces. But one tool may provide compatibility across a range of devices. A user should be trained on the tool prior to use, as a flasher may have functions that are potentially harmful to forensic investigations. Some flashers may not make a full forensic copy of the flash memory. The authors provide several resources where these tools can be found and display the functionality of one particular flasher tool [11].

The second method is to use the JTAG access ports. It is the same method as described in [13].

The third method is to physically remove the flash memory chip from the printed circuit board and read the memory with a chip programmer. This method involves de-soldering the chip from the circuit board, cleaning and preparing the chip for further processing and then reading the chip with a programmer. Each step involves different alternatives based upon the chip implementation. For example, de-soldering may be accomplished with a soldering iron or a heat gun; chip preparation and cleaning may differ based upon whether the chip was removed from a Thin Small-Outline Package or from a Ball Grid Array. Advantages of physical extraction include a guarantee of data integrity and a complete forensic image can be obtained, while disadvantages include the risk of damage [11].

After physical layer data extraction, the authors propose methods to analyze the data in order to reconstruct the logical layer. This involves a reconstruction of the flash file system. The authors liberally use the term flash file system to also include the FTL. The key steps in the process are to understand how the file systems translate the logical layer to the physical layer and how to distinguish valid from invalid data. With regards to interpreting FTL data, the authors described the difficulty of retrieving proprietary data during their attempts in identifying the controller and memory chip manufacturers. Controllers were especially difficult to identify. While most memory chips clearly were identified with the manufacturer's logo, the controllers were not. The authors used the

Internet to research the controllers, but were still unable to identify all. For those chips identified, FTL documentation was rarely found [11].

Next, the authors narrow their focus to specific implementations of flash memory on cell phones. They show successful results for reconstructing file systems on Samsung, Nokia and Symbian phones. Physical layer data was extracted using both flasher tools and by physically removing the memory chips from the phones. The file systems were recreated through the use of manufacturer provided documentation, in the case of the Samsung phone, and through heuristic methods with the Nokia and Symbian phones. In addition to a reconstruction of the current file systems, the authors were able to partially recover older file system versions through header information and the understanding of physical address schemes used in versioning data. The paper concludes with a real-world evidence collection experience [11].

In the real-world experience, the authors were able to reconstruct the Smart Media format based upon available literature. They then gave an example of how to reconstruct an unknown flash file system through identifying the metadata, identifying the granularity of the file system, analyzing the spare area metadata in order to reconstruct the logical block numbers and reconstructing the logical to physical map. The authors were able to confirm their results by comparing hash totals of the blocks of a logical and physical image [11].

3. Logical Acquisition

In “Analysis of USB Flash Drives in a Virtual Environment,” the authors Derek Bem and Ewa Huebner discuss the advantages and repercussions of using a virtual machine to analyze the contents of a USB flash drive obtained in a forensic investigation. The analysis does not consider the peculiarities inherit in flash devices, such as the FTL, flash file system, erase functionality or wear leveling. The authors acquire a logical level image of the data on the flash drive through the dd function via the FTK Imager. The authors then propose a situation where a forensic investigator would be able to mount a copy of the original imaged file, search for and record evidence without consideration to

the forensic integrity of the dd file. As far as the methodology used to acquire and analyze the data, there was no distinction made between a flash drive and a magnetic device [22].

In "An overall Assessment of Mobile Internal Acquisition Tool," Gianluigi Me present the Mobile Internal Acquisition Tool (MIAT) as a tool to acquire memory from Symbian and Windows-based smart phones via the internal memory slot, as opposed to using data cables or JTAG access pads. The system only acquires logical layer data, not physical, by use of operating system APIs. The author's motivation is to establish a method of low-level data acquisition, with minimal changes to data, which is not dependent on the large amount of non-standard cable interfaces used by competing smart phone manufacturers. The tool also offers parallel acquisition and is based upon open source tools. The use of memory cards to acquire data, with the operating system as an intermediary, reduces the hardware footprint at the crime scene and makes the MIAT very portable. The methodology is to start at the file system root and copy a file directory at a time, creating an md5 hash of each chunk copied. MIAT will recover all logical file system data structures and deleted database entries, such as contacts, through APIs, but cannot recover deleted files. MIAT does not guarantee complete data integrity, as some files may be modified through the use of some APIs. The presentation compares MIAT to the Paraben Device Seizure, a proprietary tool that also utilizes logical acquisition methods. The data recovery coverage is better than the Paraben system; is equal in integrity, but is slower in acquisition time [23].

4. Remnant Data

In "Data Remanence in Flash Memory Devices," Sergei Skorobogatov proposes a method to extract remnant data from flash cells that have been erased. Remnant data is information that can be recovered from a storage media after new information has been written over old, in attempts to delete or overwrite the old information. Remnant data is most often associated with magnetic media. This is different than residual data, which is information that has been unintentionally left behind at any level of a computer system. The author provides example targets such as smartcards and microcontrollers, which

utilize a password protected boot-loader that restricts firmware updates and data access. In particular, the author is targeting NOR flash. Typically, the on-chip operating system will completely erase code and memory before uploading new code, so that a new program cannot access old keys and previously encrypted data. The process of writing to a cell, capturing electrons, will cause a gradual accumulation of electrons in the cell which the erasing cannot release. This is one cause for the limited lifetime of flash memory, as it is no longer possible to erase a cell back to the one state after a write because of the accumulation of electrons [12].

The author found that there is a difference in the threshold voltage of a cell that was programmed to zero and then erased, and a cell that had not been erased. He also found that cells that were subjected to an erase operation, that were already erased (holding a value of one), would hold a positive charge. The author's best results were taken on cells that had not been programmed and with those that had been programmed and subject to just one erase operation. The author implemented two different methods in order to measure a cell's threshold voltage. The non-invasive method involved connecting the memory chips to a test board controlled by a PC that could directly control the voltages. The semi-invasive method involved the use of a laser diode pointer to read threshold voltages. His methods will only work on a small number of chips, which are older designs. The author provides countermeasures including cycling 10–100 program/erase operations with random data before programming sensitive data; program all cells directly prior to performing an erase operation; use chips based upon newer technology, because newer hardware with higher densities will make retrieval more difficult. The author does not explain why he chose 10 to 100 operations; the number appears to be arbitrary, but he does reference methods proposed in Peter Gutmann's controversial article regarding remnant data on magnetic disk drives [29] [30] [31]. The author also acknowledges the possibility that residual data may exist at the physical layer after blocks have been mapped out, but before the blocks have been erased [12].

In "Ten Good Reasons Why You should Shift Focus to Small Scale Digital Device Forensics," Ronald van der Knijff presents the reasons why more consideration is needed in the field of Small Scale Digital Device Forensics. The presentation is

European focused, in particular from the perspective of a member of the Netherlands Forensic Institute. His arguments include: a 100% mobile phone penetration rate in the Netherlands in 2007; personal devices are only getting smaller with flash EEPROM the most popular means to store data; flash has more potential forensic opportunity than disk drives; anti-forensic methods are more difficult on small-scale digital devices than on personal computers; the tools and procedures used for small scale devices are not as developed as other forensic tools. After presenting his argument, the author presents and evaluates various current data extraction tools and methods. At the physical level, he presents the JTAG method and the physical extraction of memory chips. At the logical level, he reviews flasher tools. In general, advantages of physical extraction include data integrity and the possibility of producing a complete forensic image. Flasher tools are easier to use, but may not guarantee a complete forensic image of the flash memory. The author presents experiences in data recovery methodology, including recreating the file system from physical layer acquisition to recreating deleted or incomplete video files. He concludes with future research opportunities [25].

In “A Study of Information Privacy and Data Sanitization Problems,” Demetrios Roubos, et al. describe the privacy issues caused by not sanitizing digital storage medium properly upon disposal. The paper provides anecdotal evidence of privacy data recovered from hard disk drives after purchase on the secondary market without the use of proper sanitization techniques. After offering a history of hard disk drives, the authors provide an overview of sanitization tools and standards. While the majority of the paper is concerned with hard disk drives, the authors provide a section on flash memory devices. A readily available tool for sanitization, such as provided for hard disk drives, is not presented; the authors provide a standard to guide the reader based upon the Department of Defense standard for sanitizing Flash EPROM. The authors guide the reader to erase the entire chip and then overwrite all locations that contained data with a character, then its compliment, and then with a random character, which is the DoD standard, 5220.22–M, "National Industrial Security Program Operating Manual" [32], and may have been supplemented by NIST SP800–88. This only addresses the logical layer, as controller proprietary commands are needed to access the physical layer data. The three pass

method may not effectively erase a flash device as spare area metadata may not be reached and artifacts created by wear leveling, out of place writes and block recovery operations may not be addressed by erasing and overwriting data at the logical layer [24].

In “An Integrated Approach to Recovering Deleted Files from NAND Flash Data,” the authors, James Luck and Mark Stokes, propose a methodology to recover deleted or corrupted MPEG-4 video through the use of recovered file metadata. The authors do not propose how the physical image was obtained, but do step through the process of rebuilding the FAT volume by building a version table containing all available versions of logical sectors. Different versions of video files can then be constructed using different versions of the same logical sectors and any missing sectors are filled with null place holders so video files may still load. The authors also used the Volume Boot Record to aid in rebuilding the FAT. For example, the VBR was able to tell the authors the sector size, the number of sectors, and the root directory. The authors make particular note about the fact that files on flash will become especially fragmented. It is not clear if they are referring to physical layer fragmentation or logical layer fragmentation, but because the authors assume that the File Allocation Table is unfragmented, they appear to be performing a logical layer recovery. The process from here entails rebuilding deleted files at the logical level and is not flash specific. The process involves recovering files through analysis of the File Allocation Table, identifying MPEG-4 data through header analysis and filling gaps within the deleted file, which reestablishes the ability to play the files on readily available video playback software [26].

In “Recovering data from USB Flash memory sticks that have been damaged or electronically erased,” Braden Phillips, Duwayne Schmidt and Dan Kelly describe a series of experiments that attempt to physically damage flash memory devices and then recover the data previously stored upon the devices. The authors used two different methods to recover the data: through connecting the flash memory device to a computer, and by directly connecting a microcontroller to the flash memory chips [27].

The authors’ experiments consisted of saving a text file and compressed audio files to a number of flash devices, and then applying over-voltage from a car battery to the signal lines, power and ground pins and to the data lines of a flash device; soaking a

flash device in water to induce corrosion and cause a short circuit; incinerating a device with petrol; stomping on a device with a rubber heel boot; striking a device with a hammer; shooting a device with a 9 mm pistol and cooking a device in a microwave. The authors were able to recover data from the devices that were damaged by over-voltage, stomped upon, soaked in water and incinerated. They were not able to read data back from the devices that were cooked in the microwave oven, smashed with a hammer and shot with the pistol. While the experiments were amusing, they lacked controls and had no way of evaluating the amount of damage that was necessary to render the device unusable [27].

5. Other

In "Algorithms and Data Structures for Flash Memories," Eran Gal and Sivan Toledo surveyed the U.S. Patent applications for technologies that address flash specific storage techniques in order to explain flash file systems. The authors state the inadequacy of magnetic disk file systems for use with flash memory, such as wear leveling. They present summarized versions of the algorithms and data structures from patent applications that address the peculiarities of flash memory and data structures used for application specific memory storage [2].

In “An Investigation into the Development of an Anti-Forensic Tool to Obscure USB Flash Drive Device Information on a Windows XP Platform,” Paula Thomas and Alun Morris analyze the registry key entries on a Windows XP system to find the changes made by USB storage devices and develop an Anti-Forensics tool that will delete or obscure these entries. After an introduction to the prevalent use of USB storage devices for legal purposes and in criminal activities, the authors describe the detail entries of registry keys in a Microsoft XP environment and the data stored that uniquely identifies a USB storage device, along with its vendor/manufacturer, product identification and how to find the drive letter assigned when it was last connected to the system. In addition to registry entries, the authors provide an analysis of log files that will contain the last time a particular device was connected to the system, and which drivers were required to be installed so the device could be used [28].

The ultimate goal of the analysis provided was to develop a tool that could obscure the changes made to a Windows XP system to cause delays in investigating the forensic trail. The authors developed a tool that can be stored upon a USB storage device and run automatically upon connection. The tool provides the following functions: the ability to add a fake device to the registry; the ability to delete the keys associated with a device that was registered on the system; amend the log file with a false entry and display the Modify, Access and Change times associated with USB device activity, with the option to email the results to a given email address. Future work included modifying registry keys that the authors overlooked and had not found after their prototype was developed, and developing better Anti-Forensic information that looked less suspicious to a potential forensic investigator [28].

III. OPPORTUNITES FOR RECOVERY

A. RESIDUAL DATA AS A RESULT OF OUT OF PLACE WRITES

1. Background

The physical characteristics of flash memory provide the opportunity to recover residual data. Residual data is information unintentionally left behind on computer media and can exist at any level of a computer system. Remnant data is information that can be recovered from a storage media after new information has been written to the media. An out of place write caused by an overwrite operation provides the best opportunity for the recovery of residual data in flash memory. An overwrite can be triggered by a user changing data within a file, altering the metadata associated with a file or deleting the file. These operations will all cause changes to be made to data at the logical level, but because of the physical limitations of flash memory, the changed data cannot be rewritten to the same flash page on the flash memory device. The Flash Transition Layer (FTL) or the flash file system will rewrite the changed data onto a different, clean flash page, mark the old page as dirty and queue the dirty page for garbage collection. How long the old page remains on the flash device without further alteration will be dependent upon the implementation of the flash file system, or FTL, and when the system decides it needs to reclaim blocks.

In addition to recovering deleted data, observation of the metadata tags and mapping structures associated with the residual data can be used to recreate older versions of files that still exist on the flash media, or at least show that the current version on the flash device is not the original version [11].

2. FTL

M-System's TrueFFS and Intel's FTL present the flash device to a host system as a block device utilizing a FAT file system. So, the host system will attempt to rewrite changes to a file back to the same sector. Because the host system is only working at the logical level, these changes also include the attempt by the host to delete a file, as this is just a rewrite, altering data in the FAT. The FTL will write the changed data to the first

clean flash page it finds. The FTL will write out the new data to the physical location and mark the old flash page as obsolete by altering the entry in the block allocation map. The new page may be within the same block if there is room. If not, the page will be written to a new block. The page will not be erased until the flash device is full and a flash block with dirty pages needs to be erased in order to free up space. Our literature review found no information on how the FTL chooses which block to erase when space is needed. Garbage collection priority is not specified within the patent, but might be based upon the number of invalid pages within the block and the block's erase count. Either way, the length of time residual data exists on the flash is not solely dependent upon the extent of utilization [33].

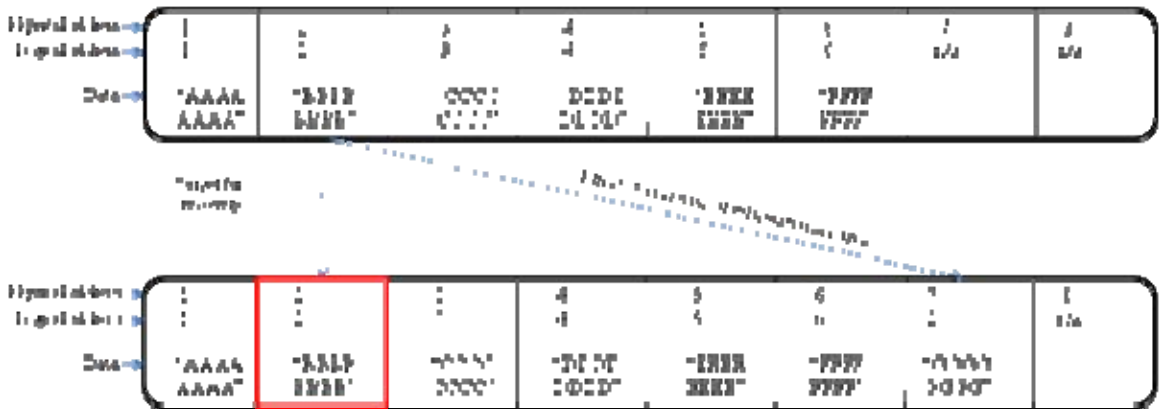

Figure 7. Target for Recovery in the FTL

TrueFFS has an additional feature called the FAT filter. TrueFFS monitors the FAT for changes that indicate clusters that have been freed. TrueFFS will in turn mark the associated pages as obsolete. The blocks containing these pages may now become available for garbage collection earlier than if the FAT filter were not implemented. This may decrease the time residual data exists on a flash memory device utilizing TrueFFS [34].

Locating obsolete pages with the same logical sequence number as valid pages may be used for file versioning. But one would have to use file carving techniques to determine if a recreated file, using an obsolete page with the same logical sequence number, makes sense, because unlike flash file systems, the FTL does not store useful

metadata such as inodes and file names in the spare area; carving would be done on 512 byte page boundaries. The only helpful information used for file carving is the file metadata stored within the data area used by the FAT file system [11].

3. YAFFS

In YAFFS, when data is overwritten in flash, the new data is written to a new flash page with the same metadata tags and the old page is simply marked as discarded. This is done by programming a data status field within the YAFFS metadata that are stored in the spare area for the flash page. But the original data within the page remains unaltered. Pages are not assigned to a dirty list and queued for garbage collection. Only blocks are assigned and only those blocks that contain one or less valid pages. If the embedded flash device has been filled and there is a need for a clean block to write additional data, YAFFS will first check the dirty list, choose a block, erase it and then write the new data. If there are no entirely dirty blocks to choose, YAFFS will choose the block with the most dirty pages, rewrite the valid pages onto a clean block, erase the block that now contains only dirty pages and write the new data. In either case, the old residual data will now be lost. The time before a page is erased will depend upon how much the flash device is utilized. The best opportunity for data recovery exists when a flash device has been written to infrequently [20].

YAFFS uses a **file id** stored in the spare area, which is similar to an inode number. A page that has been marked as invalid, but contains the same file id as a currently valid page will hold previous version data, while the page id will tell where the information belongs within the file [20].

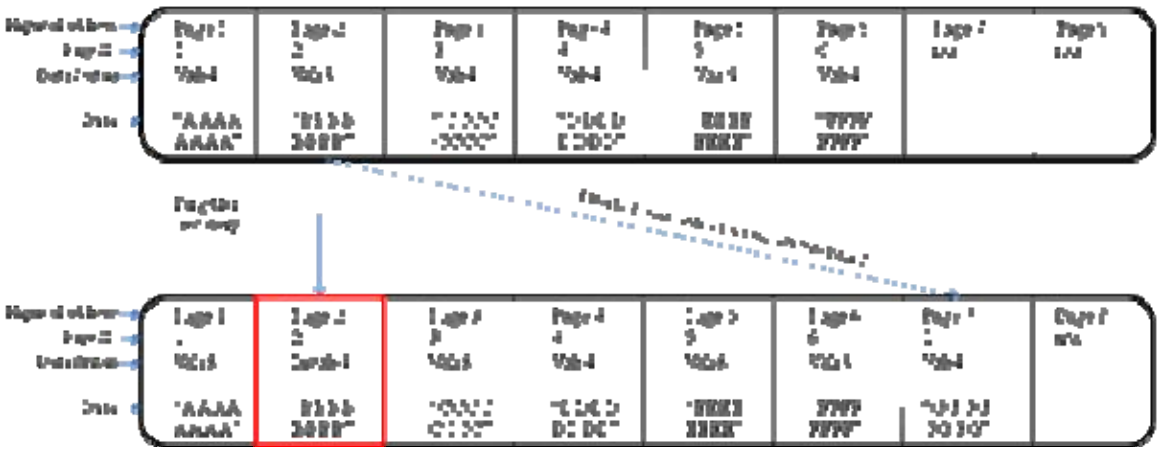

Figure 8. Target for Recovery in YAFFS

4. JFFS

The log structure used in JFFS will affect the timing of when a page is marked obsolete by an update. This log file may provide additional opportunities for the recovery of residual data. An update to a file may not completely invalidate an old page. Take the following example: 200 bytes of "A" are written to the file at offset zero. Then 200 bytes of "B" are written to the file at offset 200. Now, an update to the file is made by writing 50 bytes of "C" at offset 175. This is an overwrite to the file at offset 175 through 225. But, the first two pages will not be invalidated because page one contains valid data for offsets 0 through 175, and page two contains valid data at 225 through 400. A page is not marked dirty until a later page completely invalidates an older page. In the earlier example, if a user were to write 200 bytes of "C" at offset zero, then page one in the file would be marked as dirty. Pages are written out sequentially to the flash device until it is filled. At this point, a block will need to be reclaimed so new data may be written and the garbage collection process begins. At this point, residual data will be at risk of being erased [19].

So there are two opportunities for residual data to exist on a flash memory device using a flash file system. The first is when a page is not completely invalidated; as in the first example above. The data that exists in just the areas invalidated by a future write will contain information related to older versions of the file. Additionally, this page is

not subject to garbage collection, as it still contains valid data, and it will exist on the flash device for a longer time than if the page were invalidated.

The second opportunity exists when a page is completely invalidated, but the head of the log has not yet reached the block containing the obsolete page. Once the head of the log reaches this block, all valid pages will be written out to a clean block, and this dirty block will be erased. The length of time the page remains on the flash device will depend upon usage. A flash device utilizing JFFS, which has been written to minimally, will provide the best opportunity for the recovery of residual data.

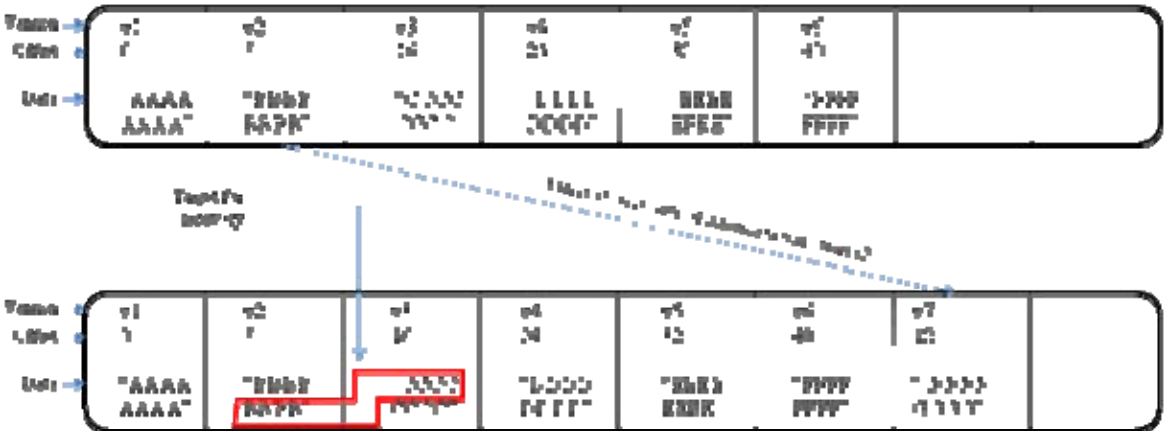

Figure 9. Target for Recovery in JFFS

Another versioning opportunity exists by examining the metadata stored with each page or node. Each node stores the parent's inode, file name and metadata in the flash page's spare area. By matching the parent inode of an obsolete page to a current in-use inode, along with the file name, an earlier version of the file can be created by inserting the data at the appropriate offset. In addition, MAC times stored with the node can also help in recreating timelines [19].

5. JFFS2

The rework for JFFS2 slightly alters the opportunity to recover residual data. All erase blocks will belong to one of three lists: a clean list that contains all blocks that have only valid pages; a dirty list that contains all blocks with at least one dirty page; and a free list that contains all blocks that contain no data. When a fresh block is needed to

write data, 99 times out of 100 the garbage collection process will choose a block from the dirty list to erase and the remaining times will choose a block from the clean list. It is not a pure log system, but it may keep residual data on the flash slightly longer, as a block with valid data is subject to garbage collection 1 out of 100 times. The versioning opportunity, for both pages remaining valid after an update and pages invalidated, remains the same as in JFFS and metadata still exists in the spare area. The opportunity for recovery is greatest on systems that are used infrequently. It is also important to note that JFFS2 uses a compression scheme, so any data recovered from the raw device will need to be uncompressed before it can be used [19].

B. EFFECTS OF FRAGMENTATION ON DATA RECOVERY

After a flash device is filled for the first time, where the flash file system or FTL decides to write new data is no longer based upon location, but will depend on variables such as the number of dirty pages in a block, or a block's erase count. Whatever method is used, a file's pages will no longer be grouped based upon spatial locality, causing files to become ever more fragmented as the flash is used. This fragmentation has two impacts.

The first impact is that larger files will become increasingly harder to recover, the more a flash device is used. Although, logically, large video and picture files will seem to have very little fragmentation, if any at all, physically they will be saved throughout the flash device as pages become available. The files will become ever more fragmented as they are altered by the user. The best opportunity of recovery is of files that can fit within a single page, or files under 512 bytes, as carving files with multiple fragmentation points is difficult [35]. This may limit recoverable items to smaller text files and lower resolution images.

The second implication of fragmentation is that it provides a signature for a flash device that has been used previously. A hard disk drive may be wiped clean, then reformatted with a new file system and can look much like a drive that was just shipped

by the manufacturer. But without physical layer access, any attempt by a user to wipe a flash memory device and then reformat it will still leave the device fragmented and might leave blocks that have not been garbage collected.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RECOVERY EXPERIMENTS

The following experiments test the theories of data recovery proposed in Chapter III. They test the effects on flash memory of a deleted file, an overwritten file and the renaming of a file. In addition, the effects of heavy usage on fragmentation is observed and how background processes, such as garbage collection and wear leveling may affect the forensic integrity of a device. The following experiments were conducted on a Linux operating system and using a NAND flash simulator mounted with the YAFFS2 and JFFS2 file systems because of their relevance. The FTL was not tested because we lacked the access to a physical device.

A. PREPARATION OF SYSTEM

The Ubuntu Linux kernel version 2.6.28 includes the JFFS2 file system. No further source code needed to be downloaded and included in the kernel source code; YAFFS2 is not included. In addition, JFFS2 has compression enabled and does not support NAND flash by default. To easily observe the effects of wear leveling techniques, JFFS2 needed to be reconfigured to turn the compression algorithms off. This allowed us to observe the effects JFFS2 had upon the physical layer, without having to research the compression algorithm and uncompress the data each time we took a physical layer image. These modifications required the kernel to be recompiled. We did not have access to a raw flash memory device (one without an FTL), so experiments were run using a simulated NAND flash device in RAM.

The following steps were taken to prepare an Ubuntu operating system in a virtual environment using VMware Workstation 6.5.0:

1. `sudo apt-get install fakeroot kernel-package libncurses5-dev linux-source-2.6.28` installed the appropriate tools for configuring and building a kernel.

2. `sudo tar xvjf /usr/src/linux-source-2.6.28.tar.bz2` uncompressed the kernel source code.

3. CVS was downloaded and installed through the synaptic package manager.

4. The YAFFS2 source code was downloaded and installed with `export CVSROOT=:pserver:anonymous@cvs.aleph1.co.uk:/home/aleph1/cvs cvs logon` and then `cvs checkout yaffs2`.

5. A change in the source code was needed in order for the kernel to compile successfully. Line 757 of file yaffs_fs.c from "pg = grab_cache_page(mapping, index);" was changed to "pg = __grab_cache_page(mapping, index);".

6. The YAFFS2 source code was copied to the appropriate location in the kernel source directory with `sudo ./patch-ker.sh c /usr/src/linux-source-2.6.28/`.

7. The commands to compile the kernel were issued from the source code directory in `/usr/src/linux-source-2.6.28/`.

8. The kernel configuration menu was launched with `sudo make menuconfig`.

9. The "Files systems" option on the configuration menu was selected first.

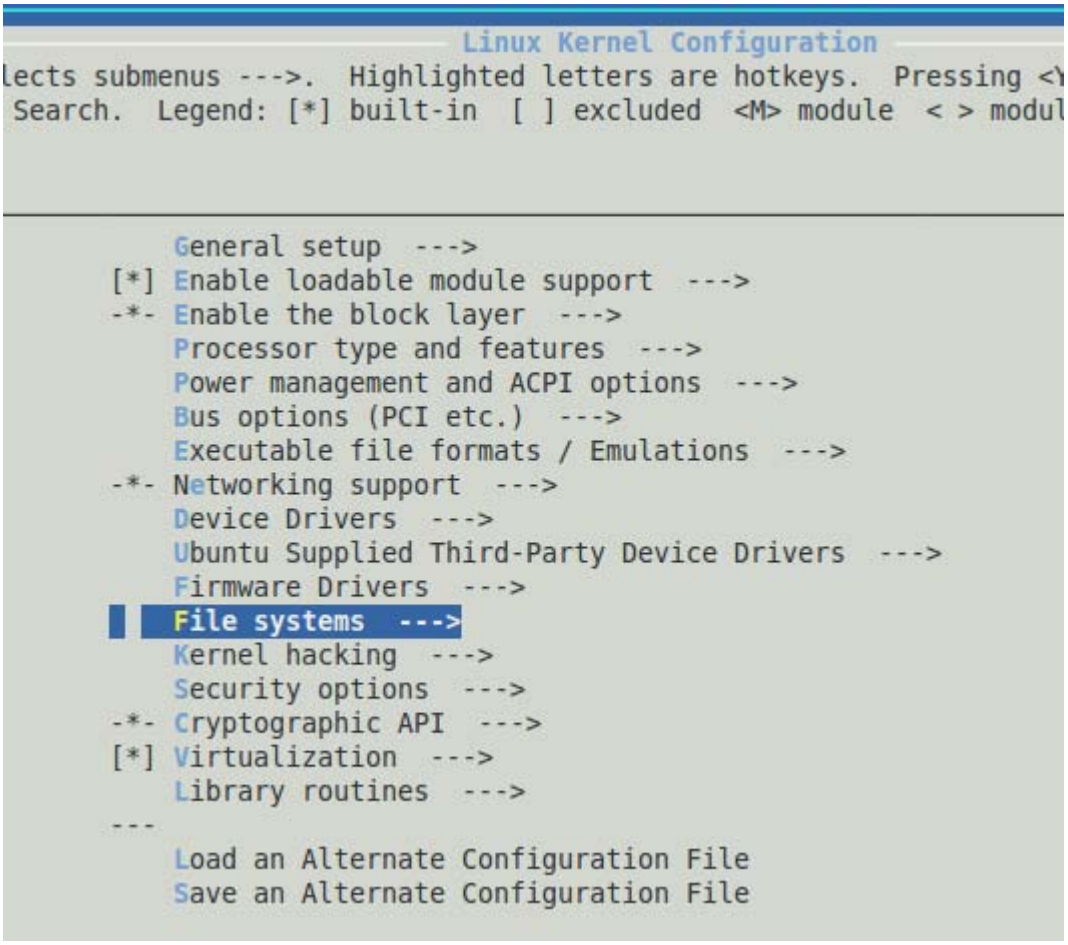

Figure 10. Kernel Configuration Step 9

10. Next, “Miscellaneous filesystems” was selected.

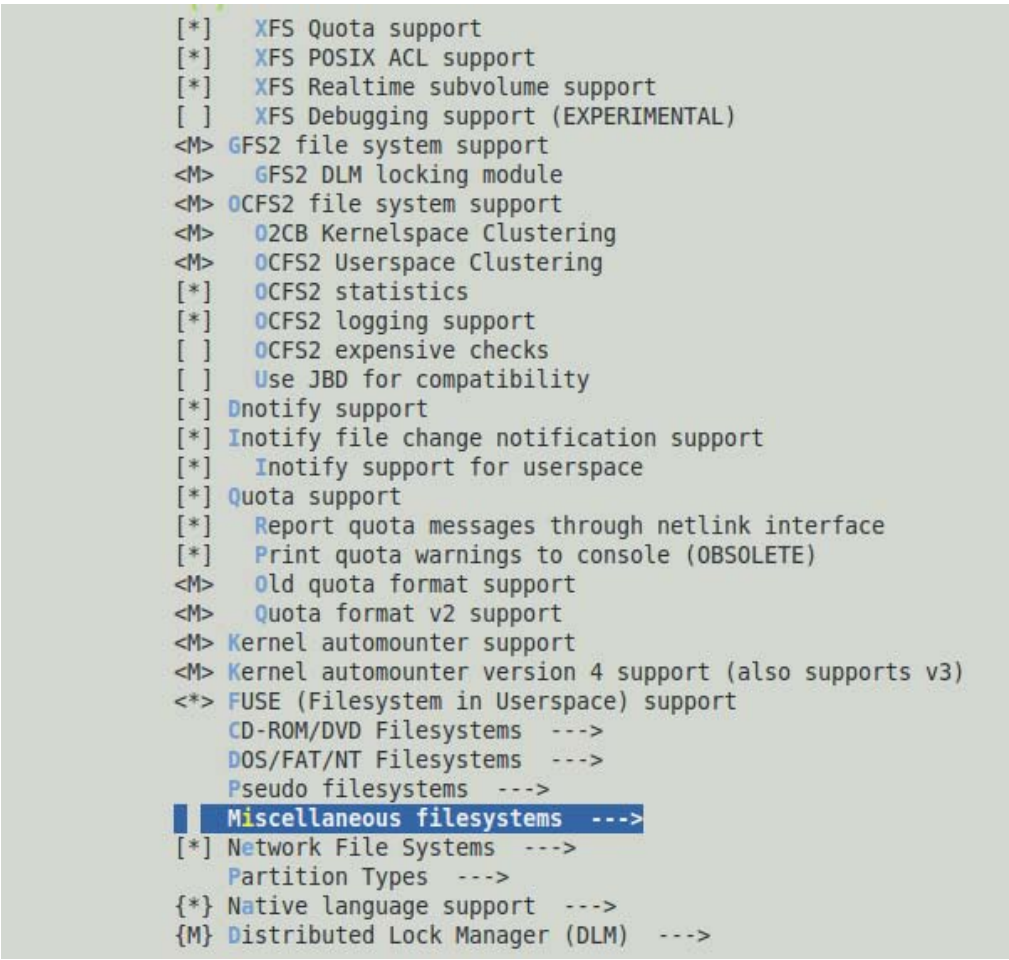

Figure 11. Kernel Configuration Step 10

11. “YAFFS2 file system support” was enabled.

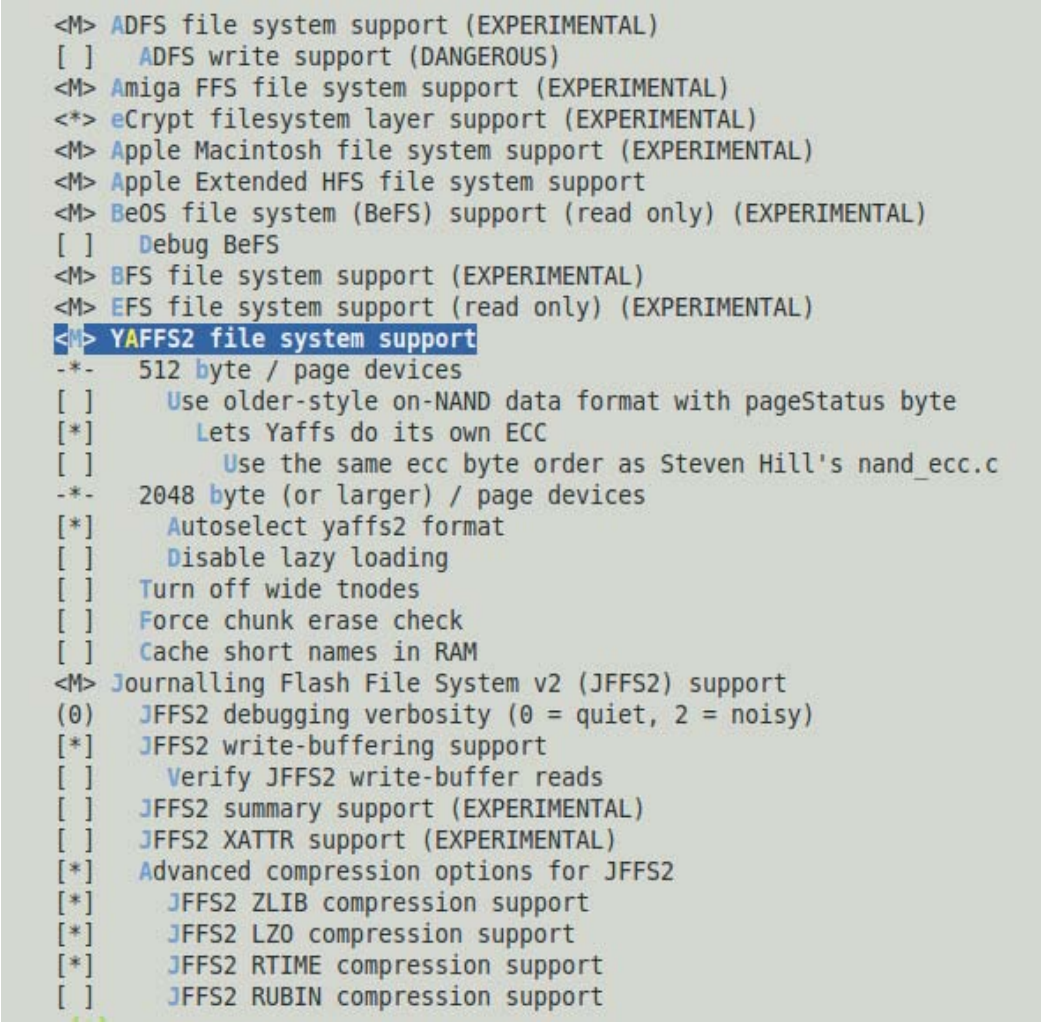

Figure 12. Kernel Configuration Step 11

12. The “Lets Yaffs do its own ECC” option was selected.

13. “Cache short names in RAM” was deselected.

14. Then, to configure JFFS2, “JFFS2 ZLIB compression support”, “JFFS2 LZO compression support” and “JFFS2 RTIME compression support” were deselected under “Journalling Flash File System v2 (JFFS2) support”. “JFFS2 write-buffering support” was left selected to provide NAND support for JFFS2.

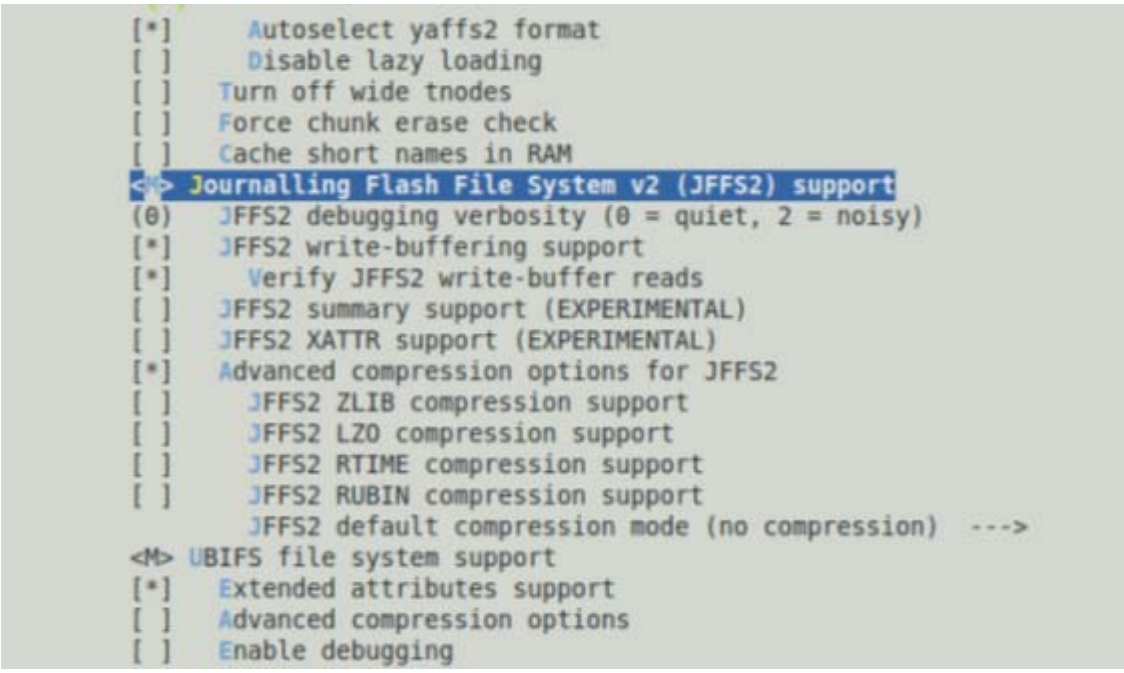

Figure 13. Kernel Configuration Step 14

15. “JFFS2 default compression mode (priority)” was selected and then “no compression” was selected to disable all compression.

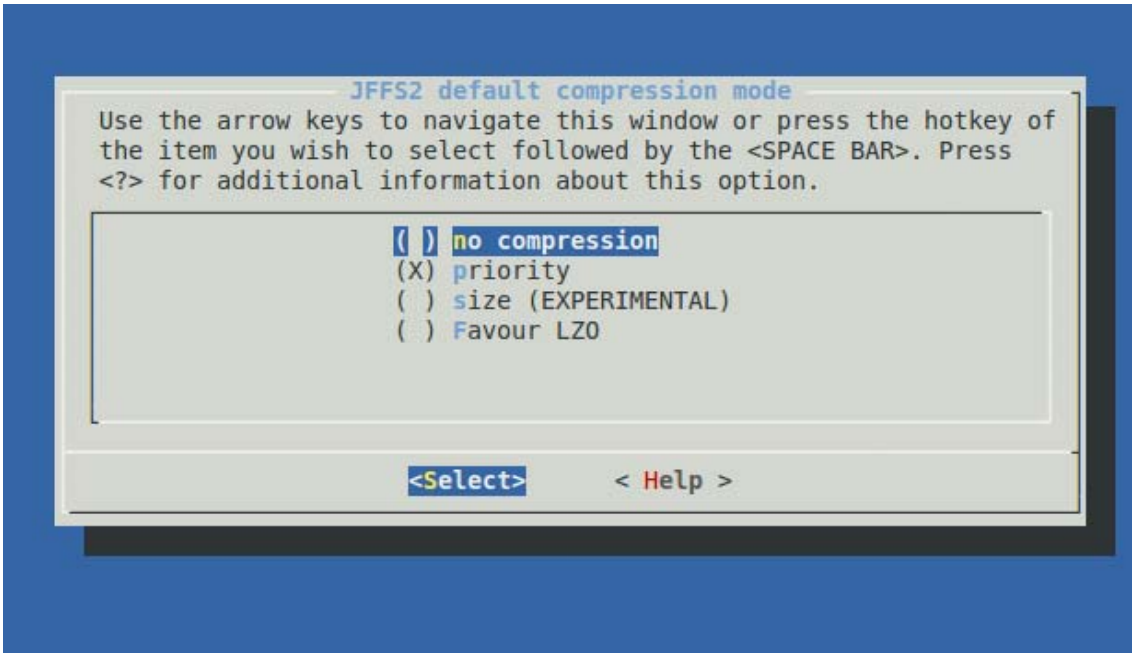

Figure 14. Kernel Configuration Step 15

16. These settings were saved and the Configuration menu was exited.

17. To clean the source tree and reset the kernel-package parameters `sudo make-kpkg clean` command was run.

18. *sudo* `fakeroot make-kpkg --initrd --revision=custom.1.0 kernel_image` compiled the kernel.

19. `sudo dpkg -i ../linux-image-2.6.28.9_custom.1.0_i386.deb` installed the new kernel.

20. `sudo shutdown -r now` rebooted the system and the new kernel loaded.

The steps to mount a JFFS2 and YAFFS2 file system onto the simulated devices were:

1. The mtd-utils package was downloaded with the Synaptic Package Manager. MTD utilities are a collection of tools that allow the user to interact with the MTD subsystem in the kernel to perform operations on Flash devices.

2. Two directories, one for JFFS2 and one for YAFFS2 to mount, were created. `/mnt/nandyaffs` and `/mnt/nandjffs` were used for these experiments.

3. `sudo modprobe mtd, modprobe jffs2, modprobe mtdchar` and `modprobe mtdblock` loaded the appropriate tools to use the mtd subsystem. `sudo modprobe nandsim first_id_byte=0x20 second_id_byte=0x33` created an mtd device of size 16MB, simulating a NAND flash device in RAM. `mtdram` and `mtd2block` can also be used to simulate a flash device, but these tools simulate NOR flash.

B. EXPERIMENTS

Each experiment was conducted on a NAND flash simulator mounted with JFFS2 and with YAFFS2. All experiments began with a blank file system. This was achieved by dismounting the mtdblock device after each experiment and then erasing the entire device. At the start of each experiment, the simulated flash device is mounted for the first time. The YAFFS2 and JFFS2 file systems were mounted on a 16MB device with 512 byte pages, 16KB erase blocks and 16 bytes of spare area. The results of the

experiments were observed by retrieving the data off of the mtd device, with the `nanddump` command, to an image file and observing the file with the hex editor hexedit. `dd`, `dcfldd` and `mtd_debug read` could all have been used to retrieve data from the mtd device, but `nanddump` gave the option to also retrieve spare area information, which best simulated a physical layer memory dump of a flash memory device. The command `sudo apt-get install hexedit` downloaded and installed hexedit. `mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs` mounted the mtd0 device with the JFFS2 file system and `mount -t yaffs2 /dev/mtdblock0 /mnt/nandyaffs` mounted the mtd0 device with the YAFFS2 file system. All experiments were conducted using Python 2.6 programs.

1. Test the Ability to Write and Read from a Simulated NAND Device

The experiment:

- The simulated flash device was mounted.
- A 512 byte file, beginning with the phrase “NPS Beginning of a page” and ending with the phrase “NPS End of a page”, was written to the flash device.
- The device was unmounted.
- The entire contents of the flash device were read and saved to an external file in order to be observed later.
- The flash device was completely erased so that a clean file system was ready for the next experiment.

The results:

The entire file containing the flag word “NPS” was recovered in both the YAFFS2 and JFFS2 mounted devices. The file was recovered on the YAFFS2 file system at byte offset 0x00. The first page was the original object header, written when the file was first created. It contained the file name and the size of the file when it was first created, which was zero. The second page contained updated object header metadata

such as user id, group id and Modified/Access/Create times. This was followed by the file's data and then the fourth page, which contained another updated object header, to include the new length of the file after the data was written to flash. YAFFS2 writes the name "silly old name" when updating object header metadata such as user id and group id. This "silly old name" operation updates far more metadata than the update operation that resulted in the write of the fourth page. In the following figures, the red lines outline one data page, while the blue lines outline one spare area, which corresponds to the data page immediately preceding it.

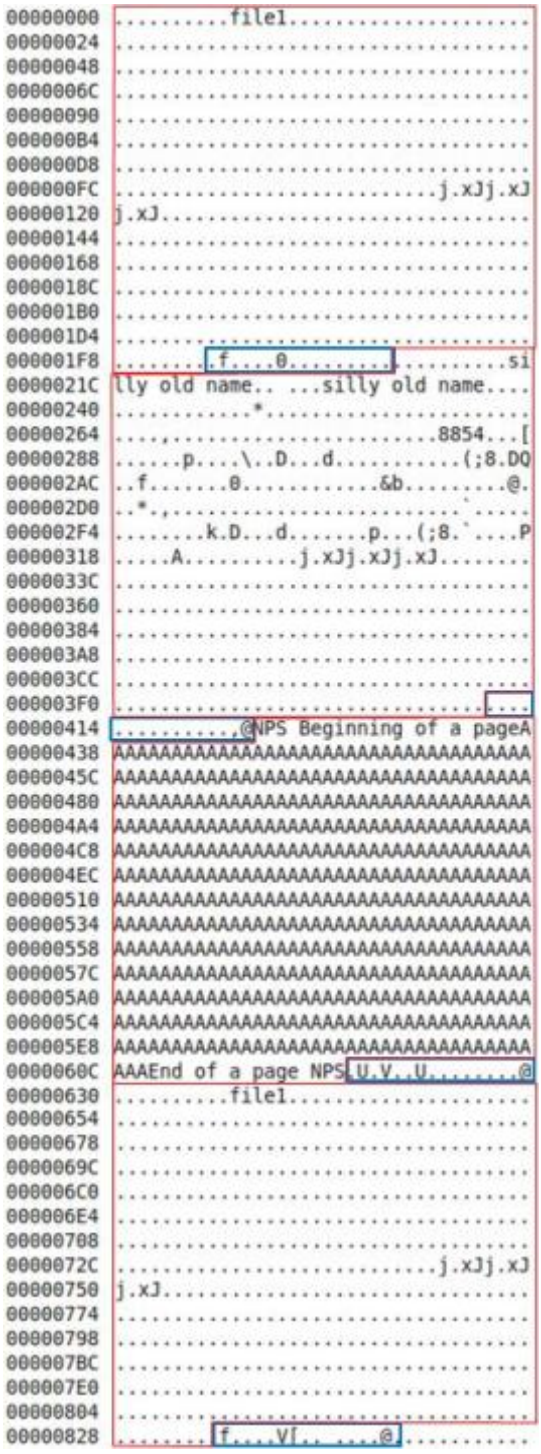

Figure 15. YAFFS2 Recovered File

The file on the JFFS2 file system was recovered at byte offset 0x1077BE8 and used two pages for both data and metadata. Metadata was stored at the beginning and the end of the file and the file's data crossed a page boundary.

Figure 16. JFFS2 Recovered File

2. Effects of Renaming a File

The experiment:

- The flash device was mounted.
- A 512 byte file named "file1", beginning with the phrase "NPS Beginning of a page" and ending with the phrase "NPS End of a page", was written to the flash device.
- The device was unmounted and the entire contents of the flash device were read and saved to an external file.

- The flash device was then mounted again and the file was renamed from “file1” to “file2”
- The device was unmounted and the entire contents of the flash device were read and saved to second external file.
- The flash device was then completely erased.

The results:

With the YAFFS2 file system, the file was recovered at byte offset 0x00. Similar to experiment 1, the first page contained the original file name and file size at the time the file was created. The second page contained the updated file header info. The third page contained the file’s data and the fourth page was an updated object header page containing the file name and new size. After the file was renamed, two additional pages were written to the flash device immediately following the first four pages, while the outdated header pages remained on the flash. The two new pages were written when the file was first touched and then when the filename was changed. The first new page contained updated object header metadata, writing the new MAC times and then rewriting information such as user id and group ids. The second page contained the new filename and another update to the MAC times. In the following image, each red block is a flash page and the subsequent blue block is its spare area.

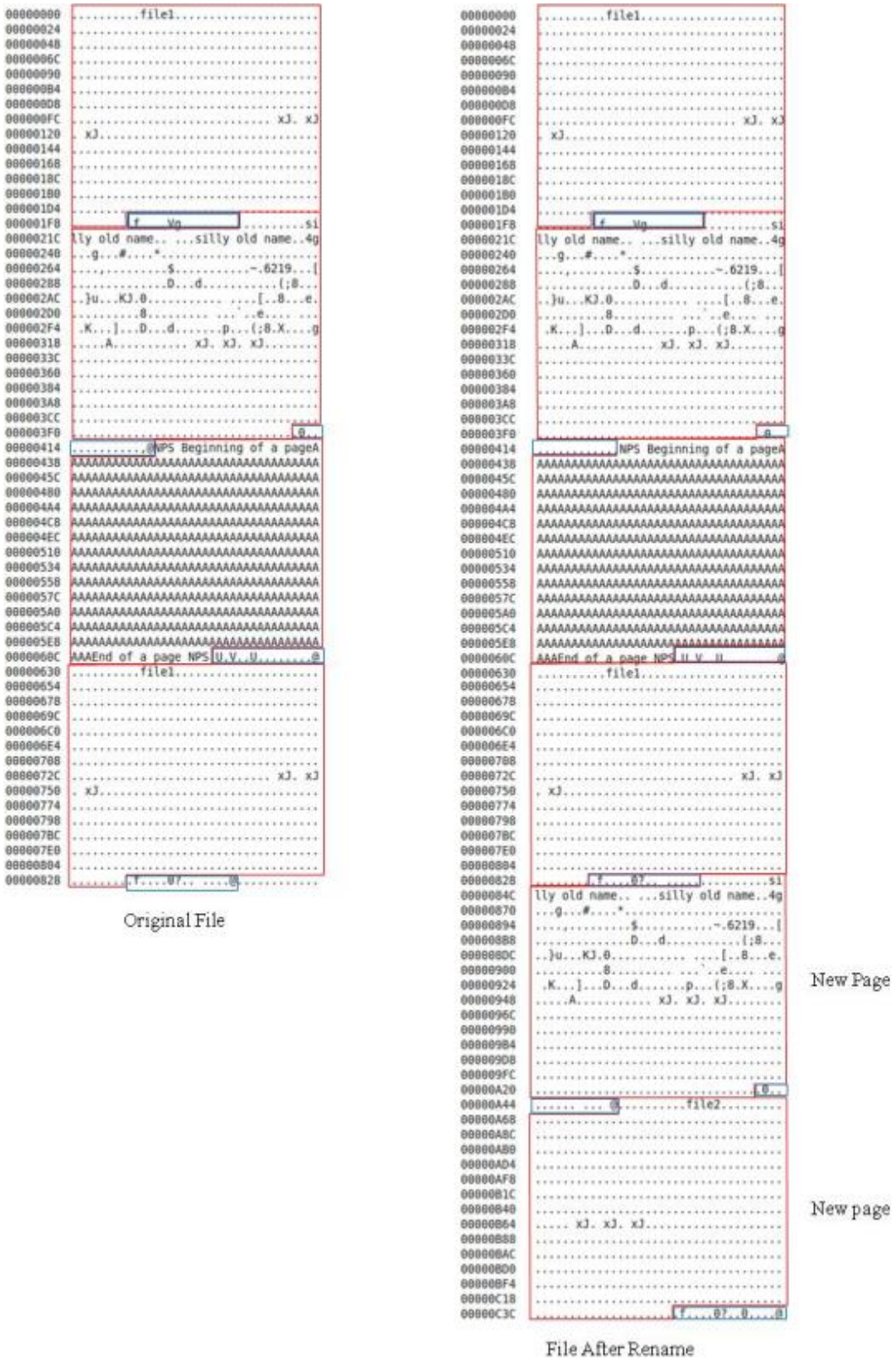

Figure 17. YAFFS2 Effects of Rename Operation

With the JFFS2 file system, the original file occupied two pages with header metadata and file data mixed into the same pages beginning at offset 0x1067400. The first page begins with metadata and the file data spanned both pages. After the rename operation, a third page was appended to the first two, containing the new header metadata, including the new filename, while the out versioned file header remained on the flash. In the following image, the red blocks represent one flash page and the blue blocks represent its associated spare area.

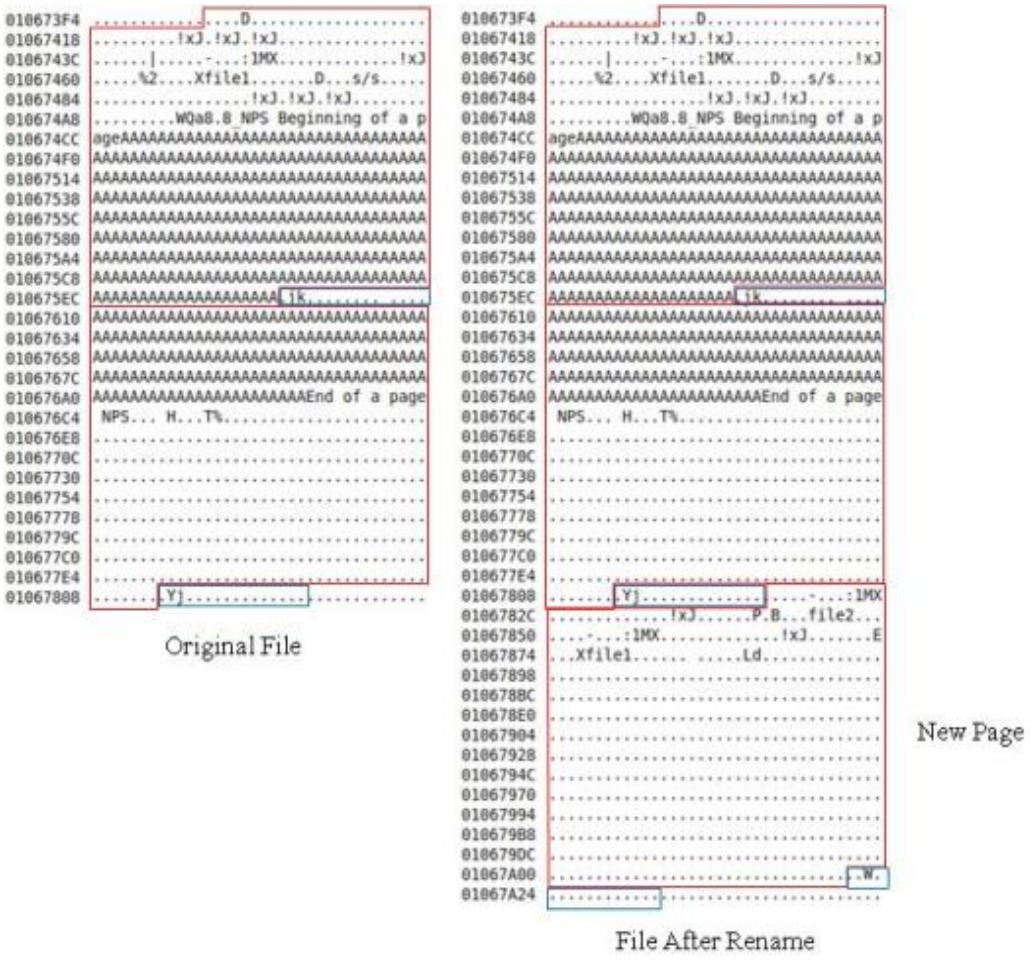

Figure 18. JFFS2 Effects of Rename Operation

3. Effects of Deleting a File

The experiment:

- The flash device was mounted; a 512 byte file named “file1”, beginning with the phrase “NPS Beginning of a page” and ending with the phrase “NPS End of a page”, was written to the flash device.
- The device was unmounted in order to flush the buffer and the entire contents of the flash device were read and saved to an external file.
- The flash device was then mounted again and the file was deleted.
- The device was unmounted and the entire contents of the flash device were read and saved to a second external file.
- The flash device was then completely erased.

The results:

With the YAFFS2 file system, the results were similar to a rename operation. Two new pages were written containing object header updates. The difference was in the second new page written as a result of the rename operation. After the file was deleted, the second updated header page contained a new filename, “unlinked,” updated MAC times, with the user id, group id and other object metadata set to zero and the size of the file reset back to zero. The original file remained on the flash device, unaltered.

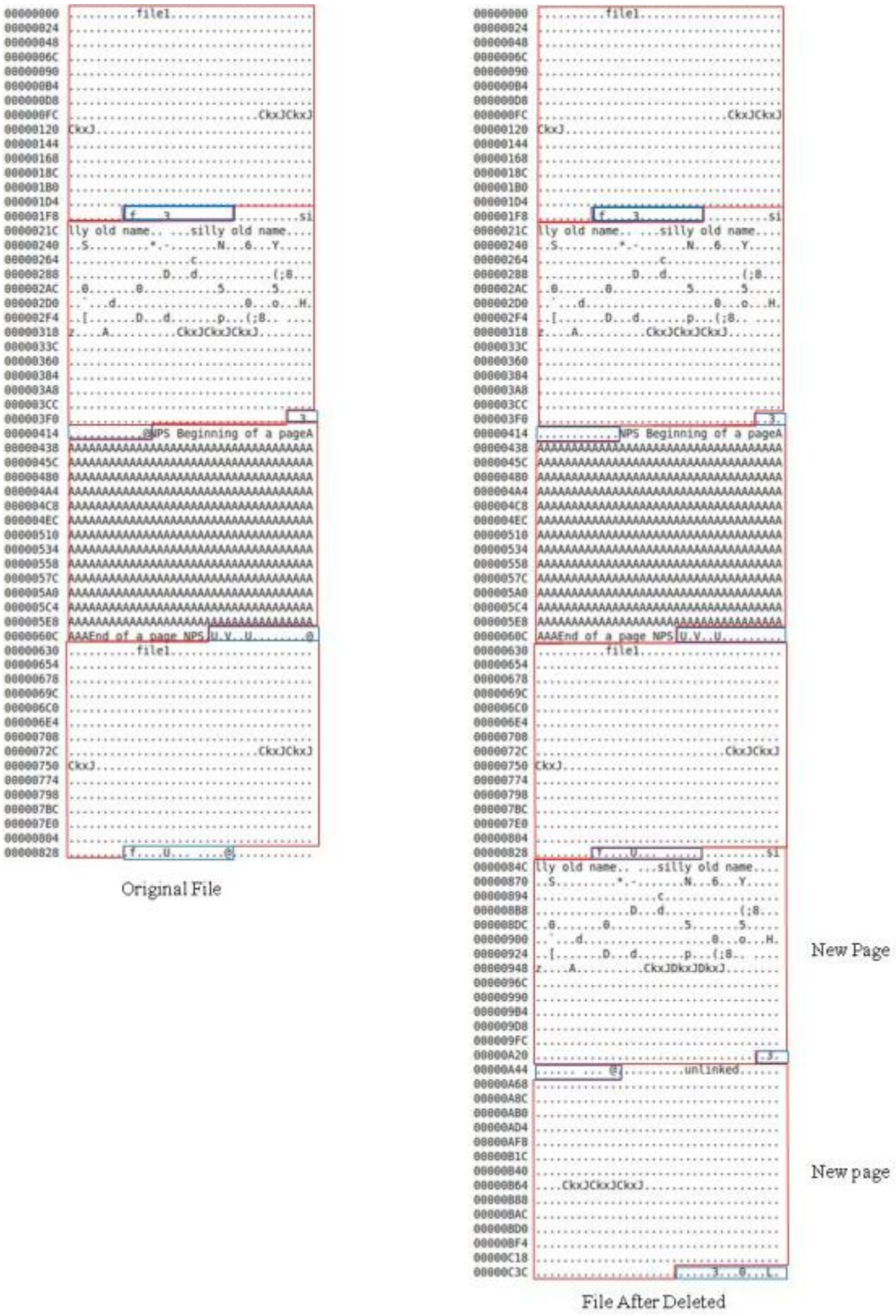

Figure 19. YAFFS2 Effects of a Delete

With the JFFS2 file system, the original file occupied two pages with header metadata and file data mixed into the same pages beginning at offset 0x1046400. The first page began with metadata and the file's data spanned both pages. After the file was deleted, a third page was written to the flash device containing updated inode information, meant to replace the original. The original file data, along with its metadata remained on the flash device, unaltered.

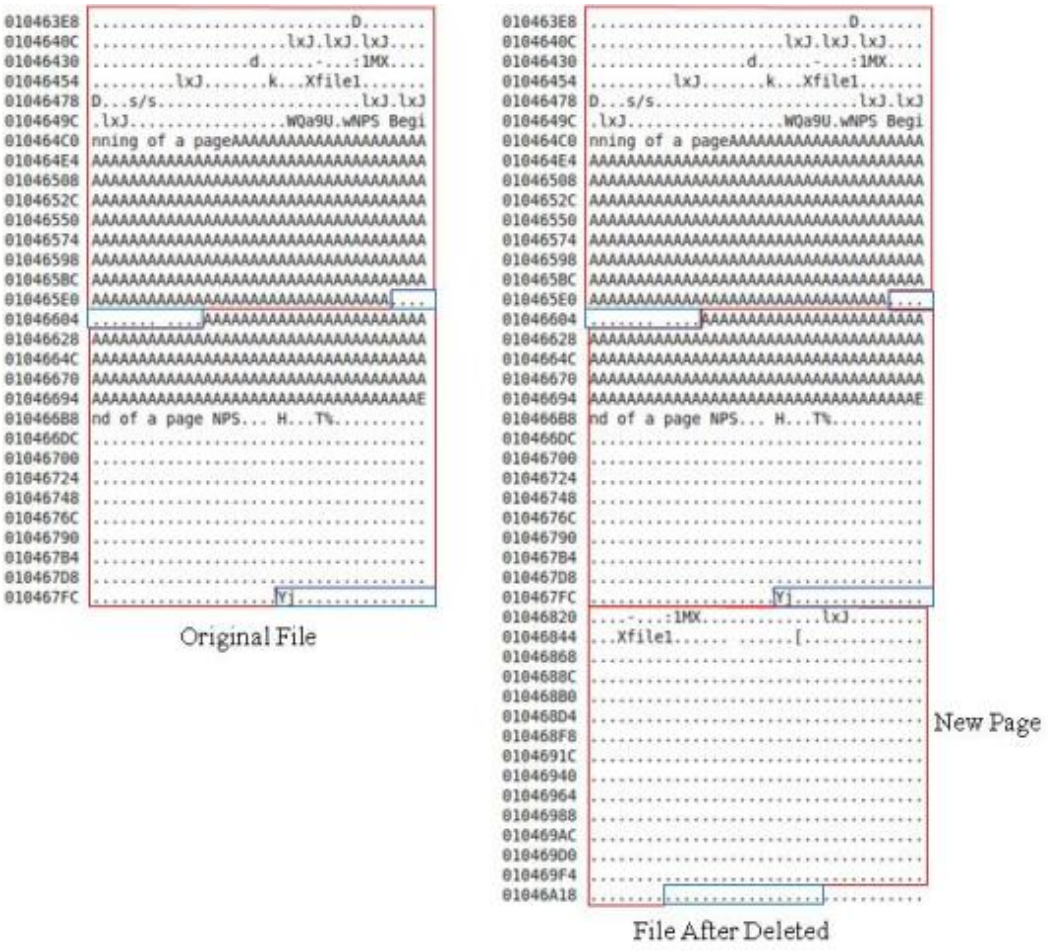

Figure 20. JFFS2 Effects of a Delete

4. Effects of a Partial Overwrite

The experiment:

- The flash device was mounted.
- A 512 byte file named “file1”, beginning with the phrase “NPS Beginning of a page” and ending with the phrase “NPS End of a page”, was written to the flash device.
- The device was unmounted in order to flush the buffer and the entire contents of the flash device were read and saved to an external file.
- The flash device was mounted again and 50 bytes of the file, at an offset 250 bytes into the file were overwritten, changing 50 ASCII “A”s to 50 ASCII “Z”s. This was accomplished through the `dd` command.
- The device was then unmounted and the entire contents of the flash device were read and saved to a second external file.
- The flash device was then completely erased.

The results:

With the YAFFS2 file system, after the partial overwrite, two new pages were written to the flash. The first page contained the entirety of the updated file’s data, including both the overwritten portion and the data that was not altered. A second page was written containing updated object header metadata including MAC times and file length, but not the robust object header update seen when “silly old name” is written. The original file remained on the device unaltered.

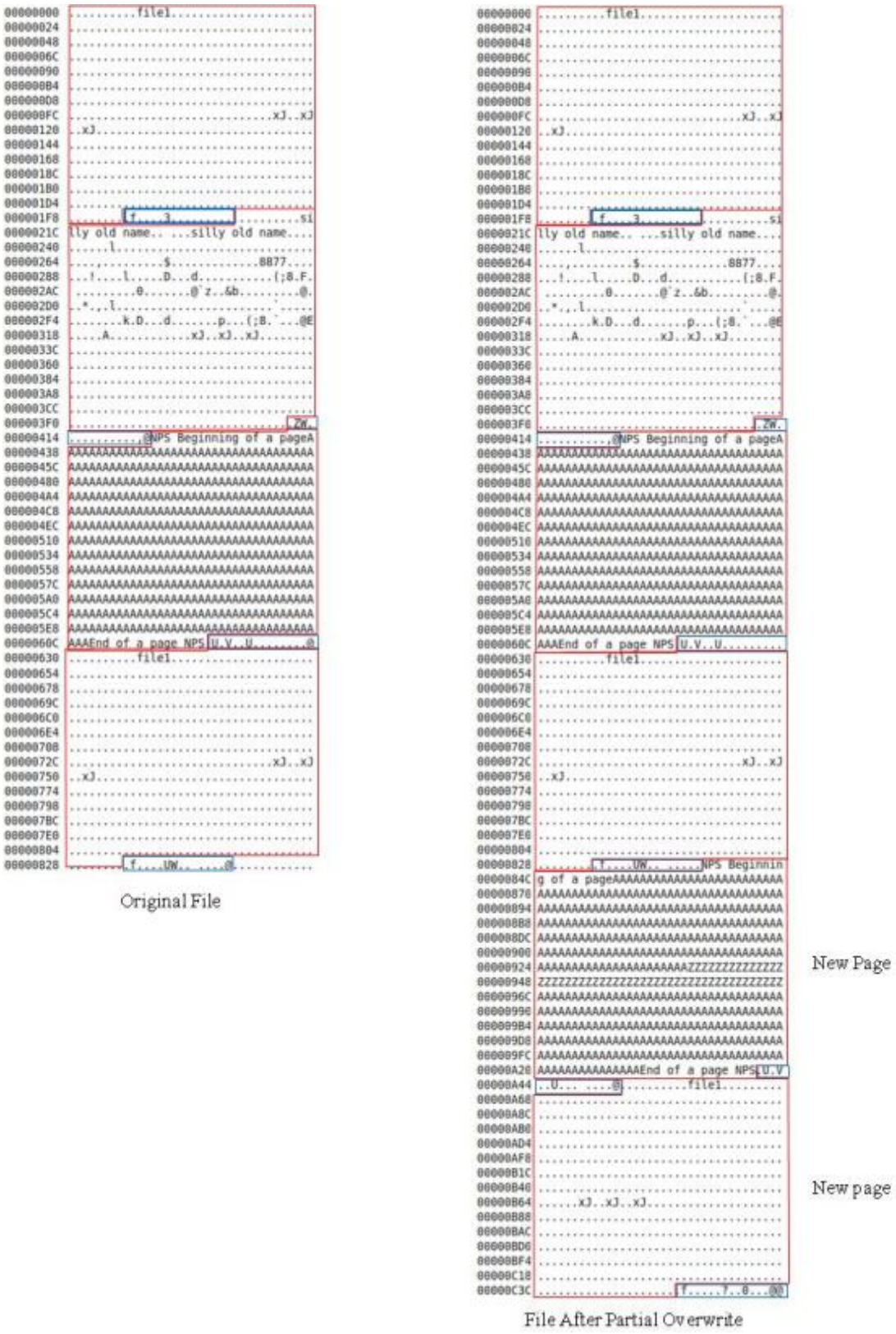

Figure 21. YAFFS2 Effects of a Partial Overwrite

With the JFFS2 file system, the new data alone was written to a new page, along with its metadata, after the original file. When the file system reads this file, it will overwrite the original file with the updated data at the appropriate offset into the file. The original file remained on the flash device, unaltered.

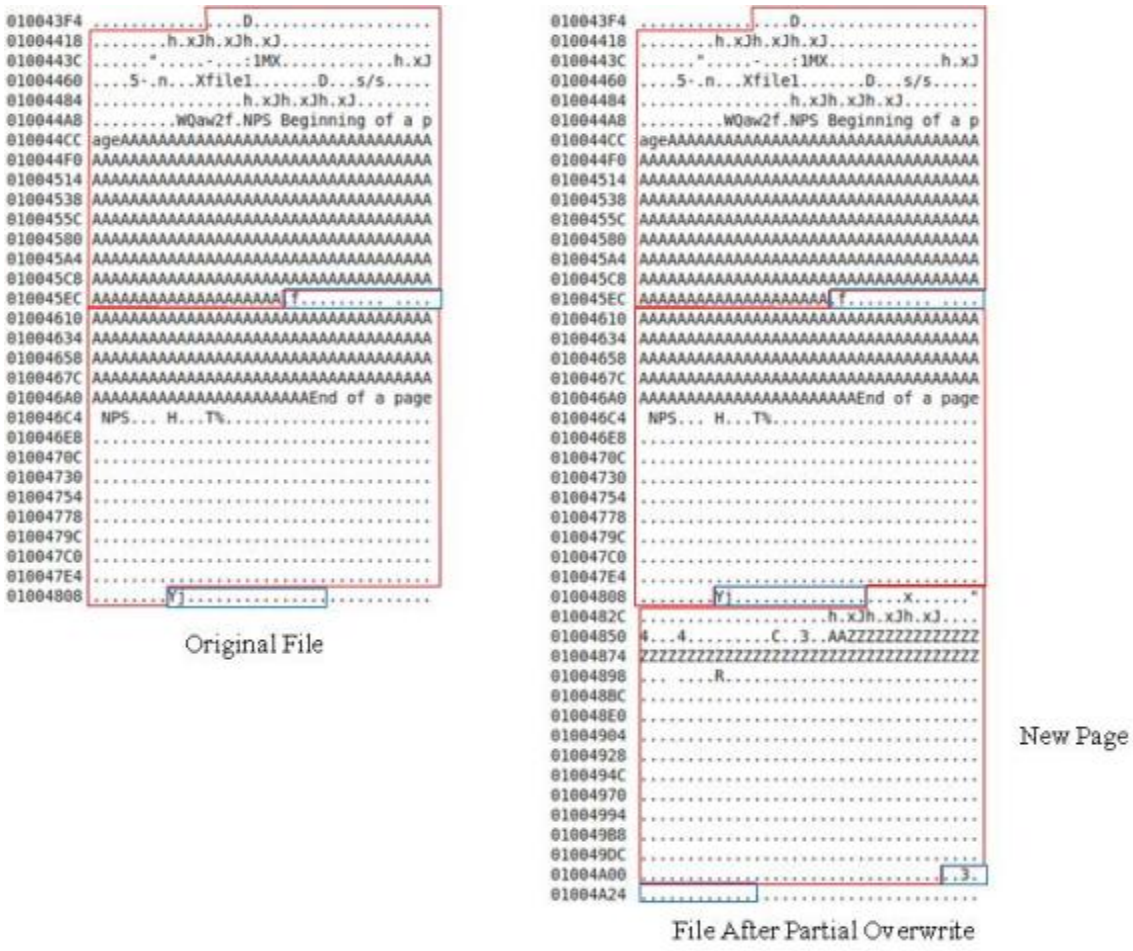

Figure 22. JFFS2 Effects of a Partial Overwrite

5. Effects of a Complete Overwrite

The experiment:

- The flash device was mounted.
- A 512 byte file named “file1”, beginning with the phrase “NPS Beginning of a page” and ending with the phrase “NPS End of a page”, was written to the flash device.
- The device was unmounted in order to flush the buffer and the entire contents of the flash device were read and saved to an external file.
- The flash device was mounted again and 512 bytes of the file, at an offset 0 bytes into the file were overwritten, changing all file contents to 512 ASCII “Z”s. This was accomplished through the `dd` command.
- The device was then unmounted and the entire contents of the flash device were read and saved to a second external file.
- The flash device was then completely erased.

The results:

With the YAFFS2 file system, after a complete overwrite, two new pages were written to the flash. The first page contained the entirety of the updated file’s data, while the second page written contained the updated object header metadata including MAC times and file length, but not the robust object header update seen when “silly old name” is written. The original file remained on the device unaltered.

Figure 23. YAFFS2 Effects of a Complete Overwrite

With the JFFS2 file system, the new data was written over two new pages, along with updated metadata, after the original file. The original file remained on the flash device, unaltered.

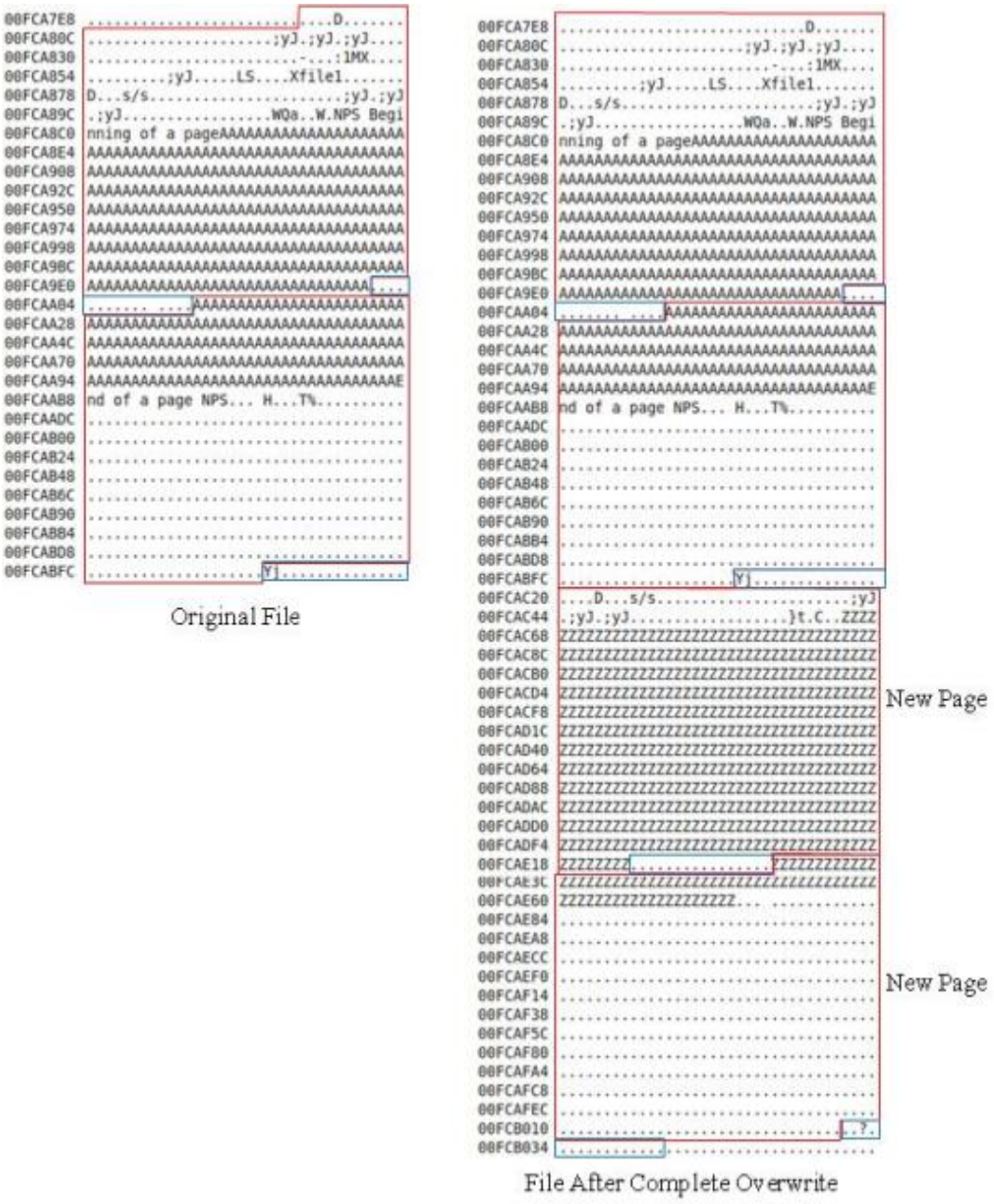

Figure 24. JFFS2 Effects of a Complete Overwrite

6. Is One File Big Enough to Sanitize?

This experiment tests the hypothesis that, if a file the size of the flash device is written, it will successfully sanitize the device of all previously saved data.

The experiment:

- The flash device was mounted.
- A 512 byte file named “file1”, beginning with the phrase “NPS Beginning of a page” and ending with the phrase “NPS End of a page”, was written to the flash device and the device was unmounted in order to flush the buffer.
- The flash device was then mounted again and “file1” was erased.
- The device was then unmounted and the entire contents of the flash device were read and saved to an external file.
- The device was mounted again and a 16MB file, containing all zeros and named “file2” was written to the device, with the dd command and a block size of 512 bytes.
- The device was unmounted again, to flush the buffer and the contents of the file were read and saved to a second external file.
- The device was then remounted and “file2” was deleted.
- The device was unmounted again and the contents were read and saved to a third external file.
- The flash device was then completely erased.

In both the YAFFS2 and JFFS2 file systems, no traces of the original file were recovered. The image obtained after “file2” was deleted was then compared to a clean image of the flash device. YAFFS2 erased all but 4 blocks after “file2” was deleted. Within those remaining four blocks, were the evidence of a sanitization attempt and a

delete operation, including the effects of updates to header objects, some residual file data along with its out of band data, the “silly old name” string and the “unlinked” string. The filename, “file2”, was not recoverable.

With JFFS2, there were nine blocks that were not erased as part of deleting “file2”. Those nine blocks contained file data and JFFS2 metadata. In addition, the filename “file2” was also recovered. When compared to a clean JFFS2 file image, these nine blocks of data showed evidence of a sanitization attempt. The remaining blocks that were successfully erased contained clean markers, which are also on the clean JFFS2 image. JFFS2 uses clean markers to ensure a block erase operation was correctly completed and blocks are safe to store information.

7. Background Processes Effect on Forensic Integrity

This experiment determines whether background process, such as garbage collection and wear leveling, will have an effect on the forensic integrity of a flash device, when there has been no change to user data.

The experiment:

- An image was taken of a clean, erased device.
- The device was mounted, unmounted and another image was taken.
- The device was mounted again.
- The program running the experiment slept for 60 seconds and then the device was unmounted.
- A third image was taken and the flash device was erased.
- An md5 sum was computed of all three images and the results were compared. The command line tool md5sum was used for computing the hash totals.

The results:

All three hash totals computed on the YAFFS2 file system were the same. For the JFFS2 file system, the first hash total computed on the clean device was different than the second and third. The second hash total and the third (calculated after the 60-second sleep operation) were the same. The difference between the first image and the second was due to clean markers written to each erase block.

8. The Effects of Heavy Usage on Fragmentation

This experiment tests whether a file written to a flash memory device that is heavily used will be fragmented at the physical layer. A fragmented file will not impact performance, but it will complicate forensic recovery.

The benchmarking program, PostMark, was used to simulate a heavily-used flash memory device found in the field. PostMark was designed to create a large pool of continually changing files, and to measure the transaction rates for a workload approximating a large Internet electronic mail server. PostMark generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. This file pool is of configurable size and can be located on any accessible file system. Once the pool has been created, a specified number of transactions occurs. Each transaction consists of a pair of smaller transactions: create a file or delete a file; read a file or append to a file [36].

The experiment was conducted on each file system twice. The difference between the two experiments was the number of files deleted after PostMark completed, in order to free memory so a new file could be created. The first experiment deletes a small number of files at the logical level in order to create a new file; the second deletes all files created by PostMark. The PostMark source code was altered so that it did not delete any files when the benchmarking completed. This gave us more control over the amount of space used on the device when we wrote a new file. The two experiments were run because of significantly different results observed in YAFFS2. For all experiments, there were 1,000 simultaneous files, ranging from 500 bytes to 10KB, and 50,000 transactions performed.

Experiment version one:

- The device was mounted and the postmark benchmarking program was run.
- The device was unmounted to flush the buffer and then remounted.
- 93KB of data was deleted to make room for a new file.
- The device was unmounted and an image was acquired.
- The device was mounted, a 50KB file was written, and the device was unmounted.
- A second image was taken and the flash device was erased.

The results:

- The file was recovered from the YAFFS2 flash device with six fragmentation points.
- The file was recovered from the JFFS2 flash device with two fragmentation points.

Experiment version two:

- The device was mounted and the postmark benchmarking program was run.
- The device was unmounted to flush the buffer and then mounted.
- Then all files were deleted to make room for a new file.
- The device was unmounted and an image was acquired.
- The device was mounted, a 50KB file was written, and the device was unmounted.
- A second image was taken and the flash device was erased.

The results:

The results were significantly different for the YAFFS2 device when all the files were deleted. YAFFS2 performed a large amount of garbage collection. All but one block was erased. This resulted in a file that was not fragmented. The block that was not garbage collected contained a small amount of residual data from the deleted files. In particular, the pages held updated header object information, including the strings “unlinked” and “silly old name,” but not file names.

JFFS2 did not perform as radical a garbage collection operation as YAFFS2. The file was recovered with three fragmentation points.

V. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

Flash memory devices are popular worldwide and include cell phones, mp3 players, SD cards, digital cameras and solid-state hard drives. Tools used today to analyze these devices treat flash memory much like a hard disk drive. The physical differences between hard disk drives and flash memory require forensic tools that are specifically designed to address flash memory.

The write once limitation of flash memory requires file changes to be stored in a different physical location. Also, wear leveling algorithms are implemented to prohibit flash memory blocks that contain frequently-altered data from going bad more quickly than those that hold static data. There are two methods to address these requirements: a flash file system and the Flash Transition Layer (FTL). The FTL allows flash devices to be used with unmodified legacy operating systems. It introduces a logical layer above the physical layer that hides the details of flash management from the operating system. USB thumb drives and SD cards utilize an FTL. A flash file system provides better utilization of flash storage at a somewhat higher cost. Two examples of flash file systems are YAFFS, which is used in Google's Android, and JFFS2, used in the OLPC program. The FTL and flash file system solutions both provide an opportunity to recover old data and metadata after a file is changed or deleted, and the new information is written to a new physical location.

This thesis contributed the following to the field of computer forensics:

- The first comprehensive survey of the academic literature regarding flash forensics.
- Thorough review of the FTL, flash file systems and flash memory patents with respect to the opportunities for recovering residual data.
- Clearly documented steps for configuring Linux to use YAFFS and JFFS2 with a flash simulator.
- Experiments which used a flash simulator and file system operations to determine residual data left by YAFFS and JFFS2.

- Discussed the possibilities for recovering residual data from the FTL.

Through the use of the YAFFS and JFFS2 file systems and a NAND flash simulator on a Linux operating system, we:

- Successfully recovered a deleted file.
- Successfully recovered a partially and a completely overwritten file.
- Successfully recovered the previous name of a file that was renamed.
- Determined that the background processes of YAFFS and JFFS2, such as garbage collection and wear leveling, did not affect the forensic integrity of the flash memory.
- Determined that writing one large file, the size of the flash memory device, was sufficient for sanitization purposes. But, evidence remained of the sanitization attempt, even after the file was deleted.
- Confirmed that a heavily used and practically full flash memory device resulted in a newly written file to have multiple fragmentation points, complicating file carving attempts. But with the use of YAFFS, the result of deleting all the files in order to free up room, triggered a large garbage collection procedure that erased all but one block, leaving newly created files unfragmented and decreasing the potential to recover old data. This was most likely caused by all of the pages in these blocks being marked as dirty, which prompted the garbage collection process.

B. FUTURE WORK

The recovery experiments were performed on clean devices with easily identifiable text. The data stored in the spare area relates the data in flash pages with its file. The spare area information allows the flash file systems and FTL to recreate the files on demand. Research that leads to a better understanding of how the information in the spare area identifies which file the data belongs will help in improving file carving techniques and data recovery attempts.

While we were able to determine that one large file was sufficient to sanitize a 16MB simulated device, we were not able to verify the effects of blocks gone bad in simulation. The FTL and flash file systems will mark blocks as bad after they can no longer be programmed correctly. Bad blocks may contain residual data that cannot be erased [37]. Additional research is needed to determine how the FTL and flash file systems treat bad blocks and whether there is potential for residual data.

The deletion of all the files stored on a full memory device in experiment eight (the effects of heavy usage on fragmentation) caused a large garbage collection that hurt the potential to recover residual data. Understanding when the FTL and flash file systems conduct garbage collection will help the forensic investigator understand how long residual data may exist on a seized flash memory device.

These experiments were conducted on a flash simulator because we did not have access to raw flash memory, and it allowed us to impose control on the tests. The use of the simulator hindered our ability to observe the effects of the FTL on flash memory. Research on devices such as USB thumb drives and SD cards will be able to test our proposed theories on residual data created by the FTL, while research on devices that use flash file systems (such as the T-Mobile G1) will provide confidence that the experiments translate well to the field.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] L. van Someren, "IDEAS List," March 5, 2008 Available: http://www.yaffs.net/ideas-list.

[2] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," ACM Computing Surveys, vol. 37, pp. 138–163, 2005.

[3] Bez, R., et al. "Introduction to Flash Memory." Proceedings of the IEEE 91.4 (2003): 489–502.

[4] J. Tyson, "How Flash Memory Works," 30 August 2000. Available:http://electronics.howstuffworks.com/flash-memory.htm (2000).

[5] M. Bauer, R. Alexis, G. Atwood, B. Baltar, A. Fazio, K. Frary, M. Hensel, M. Ishac, J. Javanifard, M. Landgraf, D. Leak, K. Loe, D. Mills, P. Ruby, R. Rozman, S. Sweha, S. Talreja and K. Wojciechowski, "A multilevel-cell 32 Mb flash memory," Solid-State Circuits Conference, 1995. Digest of Technical Papers. 42nd ISSCC, 1995 IEEE International, pp. 132–133, 351, 1995.

[6] Roberts, D., Kgil, T., and Mudge, T. 2009. Integrating NAND flash devices onto servers. Commun. ACM 52, 4 (Apr. 2009), 98–103.

[7] Cactus Technologies, "SLC vs. MLC NAND," Cactus Technologies Application Note: CTAN010, 2008.

[8] B. Fulford. June 24 2002). Unsung hero. Forbes.com Available: http://www.forbes.com/global/2002/0624/030.html

[9] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," IBM Journal of Research and Development, vol. 52, pp. 449–464, 2008.

[10] A. Tal, "NAND vs. NOR flash technology," February 2002 Available: http://www2.electronicproducts.com/NAND_vs_NOR_flash_technology-article-FEBMSY1-FEB2002.aspx.

[11] M. Breeuwsma, M. De Jongh, C. Klaver, R. van der Knijff and M. Roeloffs, "Forensics data recovery from flash memory," Small Scale Device Forensics Journal, vol. 1, pp. 1–17, 2007. Samsung Co., Ltd. NAND Flash Spare Area Assignment Standard. (2005).

[12] S. Skorobogatov, "Data remnants in flash memory devices," in Cryptographic Hardware and Embedded Systems-CHES 2005: 7th International Workshop, Edinburgh, UK, August 29-September 1, 2005: Proceedings, 2005, pp. 339.

[13] I. M. F. Breeuwsma, "Forensic imaging of embedded systems using JTAG (boundary-scan)," Digital Investigation, vol. 3, pp. 32–42, 2006.

[14] Samsung Electronics Co., Ltd, “NAND Flash Spare Area Assignment Standard,” April 27 2005 Available: http://www.samsung.com/global/business/semiconductor/products/flash/downloads/applicationnote/spare_assignment_standard.pdf

[15] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," Application Note 648, Intel Corporation, December 1998.

[16] Microsoft Corporation, "File Systems and Data Store Changes," April 13, 2005 Available: http://msdn.microsoft.com/en-us/library/ms834188.aspx.

[17] Microsoft Corporation, "File Systems and Data Store Changes," April 13, 2005 Available: http://msdn.microsoft.com/en-us/library/ms899154.aspx.

[18] M-Systems, "TrueFFS 6.x Software Development Kit (SDK) Quick Reference Guide," December 2003.

[19] D. Woodhouse, "Jffs2: The journalling flash file system, version 2," 2003-07–09 Available: http://sourceware.org/jffs2.

[20] Manning, C. "YAFFS Spec." (2002) Available: http://www.yaffs.net/yaffs-spec, last accessed September 2009.

[21] J. Gettys, "OLPC keynote," in Free and Open Source Software Developers' European Meeting, Université Libre de Bruxelles, Brussels, Belgium, 24 February 2007.

[22] D. Bem and E. Huebner, "Analysis of USB Flash Drives in a Virtual Environment," Small Scale Digital Device Forensics Journal, vol. 1, 2007.

[23] A. Distefano and G. Me, "An overall assessment of Mobile Internal Acquisition Tool," Digital Investigation, vol. 5, pp. 121–127, 2008.

[24] Roubos, D. Palmieri, L. Kachur, R. L. Herath, S. Herath, A. Constantino, D., "A Study of Information Privacy and Data Sanitization Problems," CCSC: South Central Conference, vol. 22, pp. 212, April 2007.

[25] R. Knijff, "Ten Good Reasons Why You Should Shift Focus to Small Scale Digital Device Forensics," URL http://www.dfrws.org/2007/proceedings/vanderknijff_pres.pdf , vol. 6, 2007, last accessed September 2009.

[26] J. Luck and M. Stokes, "An Integrated Approach to Recovering Deleted Files from NAND Flash Data".

[27] B. Phillips, C. Schmidt and D. Kelly, "Recovering data from USB flash memory sticks that have been damaged or electronically erased," in Proceedings of the 1st International Conference on Forensic Applications and Techniques in Telecommunications, Information, and Multimedia and Workshop Table of Contents, 2008.

[28] P. Thomas and A. Morris, "An investigation into the development of an anti-forensic tool to obscure USB flash drive device information on a windows XP platform," in Digital Forensics and Incident Analysis, 2008. WDFIA'08. Third International Annual Workshop on, 2008, pp. 60–66.

[29] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory," in Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography-Volume 6, 1996.

[30] D. Feenberg, “Can Intelligence Agencies Read Overwritten Data? A response to Gutmann,” National Bureau of Economic Research, 2003. http://www.nber.org/sys-admin/overwritten-data-gutmann.html

[31] C. Wright, D. Kleiman and S. S. RS, "Overwriting hard drive data: The great wiping controversy," in Proceedings of the 4th International Conference on Information Systems Security, 2008, pp. 243–257.

[32] DoD Directive, "5220.22-M-Sup 1," National Industrial Security Program Operating Manual, 1995.

[33] A. Ban, Flash File System, 1995.

[34] D. Shmidt, Technical Note: Trueffs Wear-Leveling Mechanism (Tn-Doc-017).

[35] A. Pal, H. T. Sencar and N. Memon, "Detecting file fragmentation point using sequential hypothesis testing," Digital Investigation, vol. 5, pp. 2–13, 2008.

[36] J. Katcher, PostMark: A New File System Benchmark, 1997. http://communities.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pdf , last accessed September 2009.

[37] S. L. Garfinkel and A. Shelat, "Remembrance of data passed: A study of disk sanitization practices," IEEE Security & Privacy, pp. 17–27, 2003.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX

A. PYTHON CODE FOR EXPERIMENTS

1. YAFFS Experiment 1

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t yaffs2 '+
'/dev/mtdblock0 /mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment1Image')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

2. YAFFS Experiment 2

```
#!/usr/bin/env python
import commands, os
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment2Image')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
os.rename('/mnt/nandyaffs/file1', '/mnt/nandyaffs/file2')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment2ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

3. YAFFS Experiment 3

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment3Image')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
```

```
commands.getstatusoutput('rm /mnt/nandyaffs/file1')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment3ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

4. YAFFS Experiment 4

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment4Image')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('dd if=/mnt/Z of=/mnt/nandyaffs/file1 obs=50
ibs=50 seek=5 count=1 conv=notrunc')  #/mnt/Z is a file containing 50
ASCII “Z”
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment4ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

5. YAFFS Experiment 5

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment7Image')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0
/mnt/nandyaffs')
commands.getstatusoutput('dd if=/mnt/Z3 of=/mnt/nandyaffs/file1 obs=512
ibs=512 count=1 conv=notrunc') #/mnt/Z3 is a file containing 512 ASCII
“Z”
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment7ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

6. YAFFS Experiment 6

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('umount /dev/mtdblock0')
```

```
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment5Virgin')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('rm /mnt/nandyaffs/file1')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment5Image')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('dd if=/dev/zero of=/mnt/nandyaffs/file2 bs=512
count=32768')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment5ImageAfterSanitization')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('rm /mnt/nandyaffs/file2')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment5ImageAfterDeleteFile2')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

7. YAFFS Experiment 7

```
#!/usr/bin/env python
import commands, time
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment6Image1')
commands.getstatusoutput('md5sum -b '+
'/home/user/Desktop/yaffsexp/experiment6Image1 > '+
'/home/user/Desktop/yaffsexp/experiment6MD5Sums')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('sudo umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment6Image2')
commands.getstatusoutput('md5sum -b '+
'/home/user/Desktop/yaffsexp/experiment6Image2 >> '+
'/home/user/Desktop/yaffsexp/experiment6MD5Sums')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
time.sleep(60)
commands.getstatusoutput('sudo umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment6Image3')
commands.getstatusoutput('md5sum -b '+
'/home/user/Desktop/yaffsexp/experiment6Image3  >> '+
'/home/user/Desktop/yaffsexp/experiment6MD5Sums')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

8. YAFFS Experiment 8.1

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('rm -rf /mnt/nandyaffs/26*')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment8Image')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
for x in range(0,97):
      f.write('NPS Beginning of a page')
      f.write('A'*473)
      f.write('End of a page NPS')
f.write('NPS Beginning of a page')
f.write('A'*313)
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment8ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

9. YAFFS Experiment 8.2

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
commands.getstatusoutput('rm -rf /mnt/nandyaffs/*')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment8aImage')
commands.getstatusoutput('mount -t yaffs2 /dev/mtdblock0 '+
'/mnt/nandyaffs')
f = open('/mnt/nandyaffs/file1', 'wb')
for x in range(0,97):
      f.write('NPS Beginning of a page')
      f.write('A'*473)
      f.write('End of a page NPS')
f.write('NPS Beginning of a page')
f.write('A'*313)
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/yaffsexp/experiment8aImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

10. JFFS2 Experiment 1

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
```

```
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment1Image')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

11. JFFS2 Experiment 2

```
#!/usr/bin/env python
import commands, os
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment2Image')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
os.rename('/mnt/nandjffs/file1', '/mnt/nandjffs/file2')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment2ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

12. JFFS2 Experiment 3

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment3Image')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('rm /mnt/nandjffs/file1')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment3ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

13. JFFS2 Experiment 4

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment4Image')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
```

```
commands.getstatusoutput('dd if=/mnt/Z of=/mnt/nandjffs/file1 obs=50
ibs=50 seek=5 count=1 conv=notrunc') #/mnt/Z is a file containing 50
ASCII “Z”
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment4ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

14. JFFS2 Experiment 5

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment7Image')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('dd if=/mnt/Z3 of=/mnt/nandjffs/file1 obs=512
ibs=512 count=1 conv=notrunc') #/mnt/Z3 is a file containing 512 ASCII
“Z”
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment7ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

15. JFFS2 Experiment 6

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment5Virgin')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
f.write('NPS Beginning of a page')
f.write('A'*473)
f.write('End of a page NPS')
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('rm /mnt/nandjffs/file1')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment5Image')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('dd if=/dev/zero of=/mnt/nandjffs/file2 bs=512
count=32768')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment5ImageAfterSanitization')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('rm /mnt/nandjffs/file2')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment5ImageAfterDeleteFile2')
```

```
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

16. JFFS2 Experiment 7

```
#!/usr/bin/env python
import commands, time
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment6Image1')
commands.getstatusoutput('md5sum -b '+
'/home/user/Desktop/jffsexp/experiment6Image1 >
/home/user/Desktop/jffsexp/experiment6MD5Sums')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('sudo umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 >
/home/user/Desktop/jffsexp/experiment6Image2')
commands.getstatusoutput('md5sum -b '+
'/home/user/Desktop/jffsexp/experiment6Image2 >> '+
'/home/user/Desktop/jffsexp/experiment6MD5Sums')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
time.sleep(60)
commands.getstatusoutput('sudo umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment6Image3')
commands.getstatusoutput('md5sum -b '+
'/home/user/Desktop/jffsexp/experiment6Image3  >> '+
'/home/user/Desktop/jffsexp/experiment6MD5Sums')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

17. JFFS2 Experiment 8.1

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('rm -rf /mnt/nandjffs/26*')
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment8Image')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
for x in range(0,97):
      f.write('NPS Beginning of a page')
      f.write('A'*473)
      f.write('End of a page NPS')
f.write('NPS Beginning of a page')
f.write('A'*313)
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment8ImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

18. JFFS2 Experiment 8.2

```
#!/usr/bin/env python
import commands
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
commands.getstatusoutput('rm -rf /mnt/nandjffs/*')
```

```
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment8aImage')
commands.getstatusoutput('mount -t jffs2 /dev/mtdblock0 /mnt/nandjffs')
f = open('/mnt/nandjffs/file1', 'wb')
for x in range(0,97):
      f.write('NPS Beginning of a page')
      f.write('A'*473)
      f.write('End of a page NPS')
f.write('NPS Beginning of a page')
f.write('A'*313)
f.close()
commands.getstatusoutput('umount /dev/mtdblock0')
commands.getstatusoutput('nanddump /dev/mtd0 > '+
'/home/user/Desktop/jffsexp/experiment8aImageA')
commands.getstatusoutput('mtd_debug erase /dev/mtd0 0 16777216')
```

B. LINUX IMAGE README

This is the linux kernel image we used to run our experiments for the thesis "The Forensic Potential of Flash Memory"

The username is: user

The password is: password

The kernel has been recompiled to include the YAFFS2 and JFFS2 with no compression and NAND support

The following functionality has also been loaded:

- libncurses5 (needed to recompile the kernel)

- kernel-package (needed to recompile the kernel)

- linux source code 2.6.28 (needed to recompile the kernel)

- YAFFS2 (The source code is in the home directory)

- mtd-utils

- hexedit hex editor

- PostMark benchmarking program (The original tarball and the altered source code are in the home directory)

- The mtd, jffs2, mtdchar and mtdblock modules have been loaded (If you restart the system, these will need to be reloaded)

- A clean 16MB simulated flash device with 512 byte pages and 16Kb erase block on mtd0. (If you restart the system, this simulated device will need to be recreated)

The python programs used to run the experiments are on the Desktop in the yaffsexp and jffsexp directories

The directories used to mount the simulated flash device have been created and are located at:

- /mnt/nandjffs

- /mnt/nandyaffs

If you find any sensitive/private information, please remove and contact us at: jaregan@nps.edu or slgarfin@nps.edu

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California